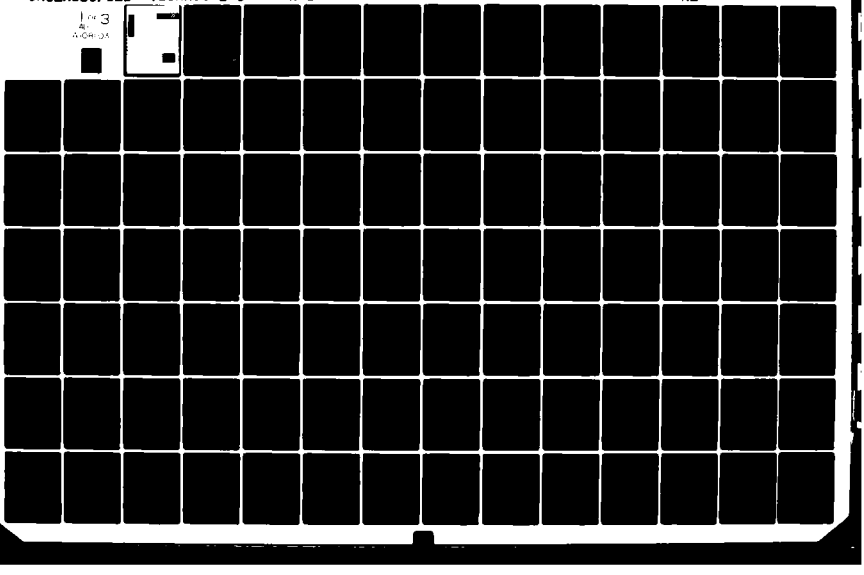
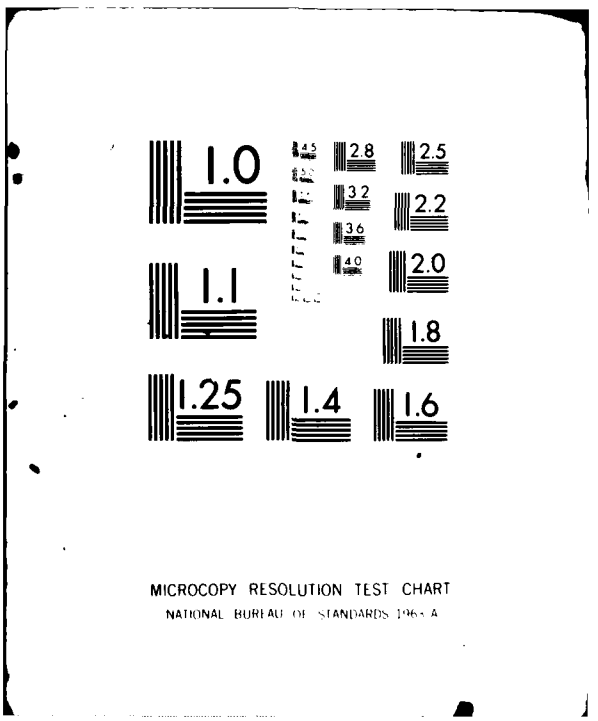
AD-A108 103 BOLT BERANEK AND NEWMAN INC CAMBRIDGE MA F/G 9/2
THE TERMINAL INTERFACE MESSAGE PROCESSOR PROGRAM.(U)
NOV 73 DAHC15-69-C-0179
UNCLASSIFIED TECHNICAL INFORMATION-91 NL




1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

BOLT BERANEK AND NEWMAN, INC.

(1)

LEVEL II

THE TERMINAL INTERFACE MESSAGE PROCESSOR PROGRAM

August 1973

Update Edition of November 1973

Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A	

11/73

Table of Contents

1. OVERVIEW OF THE TERMINAL IMP HARDWARE

Understanding the Terminal IMP software depends on an understanding of the hardware environment in which the software resides. A summary of the Terminal IMP hardware follows.

Up to 63* terminals, either remote or local, of widely diverse characteristics may be connected to a given TIP and thereby "talk" into the network. It is also possible to connect a Host to a TIP in the usual way a Host is connected to an IMP.

The TIP is built around a Honeywell H-316 computer with 28K of core. It embodies a standard 16-port multiplexed memory channel with priority interrupts and includes a Teletype for debugging and program reloading. Other features of the standard IMP also present are a real-time clock, power-fail and auto-restart mechanisms, and a program-generated interrupt feature. As in the standard IMP, interfaces are provided for connecting to high-speed (50-kilobit, 230.4-kilobit, etc.) modems as well as to Hosts.

Aside from the additional 12K of core memory, the primary hardware feature which distinguishes the TIP from a standard IMP is a Multi-Line Controller (MLC) which allows for connection of terminals to the IMP. Any of the MLC lines may go to local terminals or via modems to remote terminals. As shown in Figure 1-1 the MLC consists of two portions, one a piece of central logic which handles the assembly and disassembly of characters and transfers them to and from memory, and the other a set of individual Line Interface Units (all identical except for small number of option jumpers) which synchronize reporting to individual data bits between the central logic and the terminal devices and provide for control and status information to and from the modem or device. Line Interface Units may be physically incorporated one at a time as required.

---

*There are 64 hardware lines but line Ø is logically reserved by the program for special use.

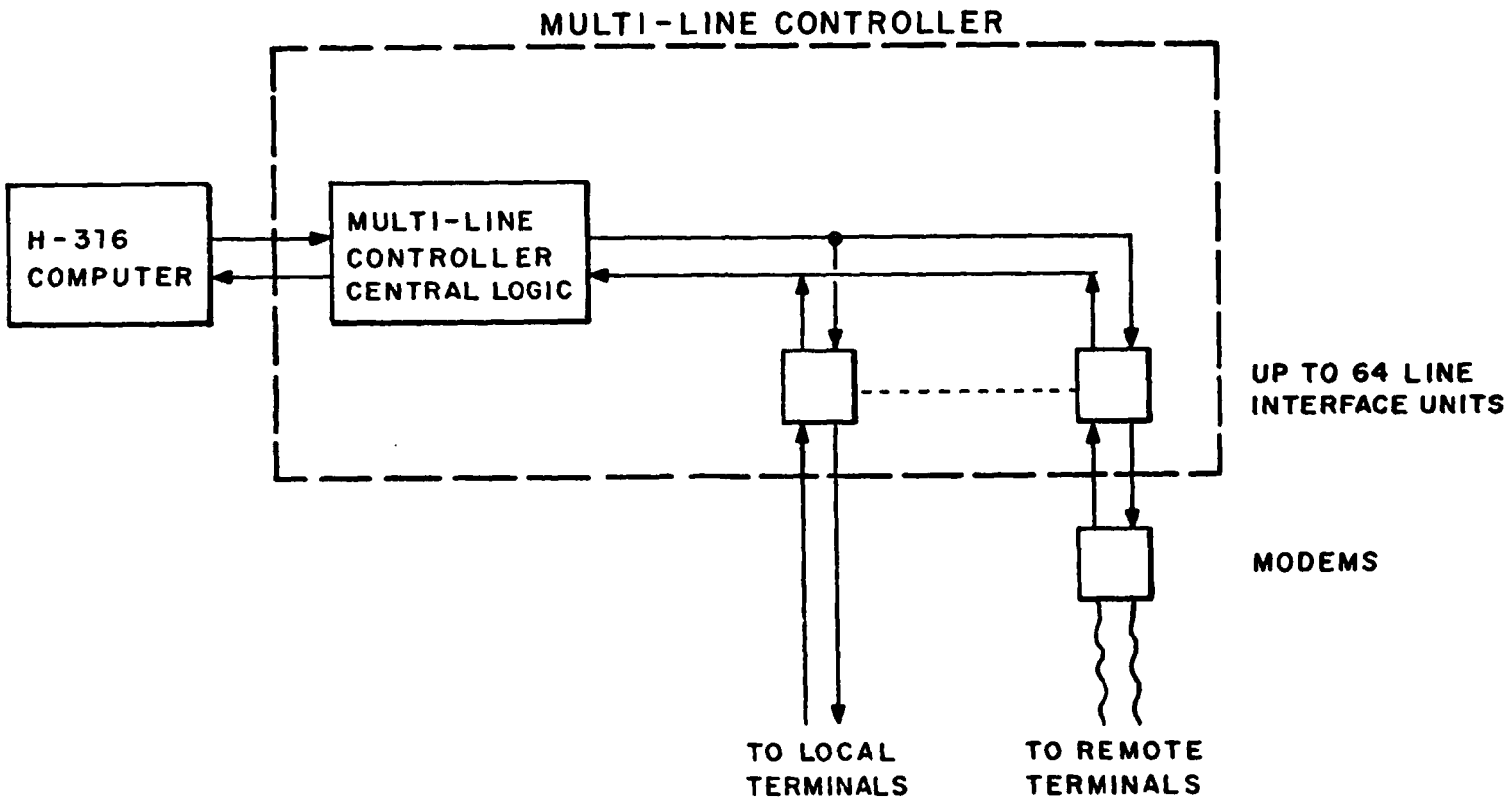


FIGURE 1-1 BLOCK DIAGRAM OF THE TIP HARDWARE

The MLC connects to the high-speed multiplexed memory channel option of the H-316, and uses three of its channels as well as two priority interrupts and a small number of control instructions.

In order to accommodate a variety of devices, the controller handles a wide range of data rates and character sizes, in both synchronous and asynchronous modes. Data characters of length 5, 6, 7, or 8 bits are allowed by the controller. Since no interpretation of characters is done by the hardware, any character set, such as ASCII or EBCDIC, may be used.

The following is a list of data rates accepted by the controller although all are not yet handled by the software.

SYNCHRONOUS	ASYNCHRONOUS (Nominal Rates)	
Any rate up to and including 19.2 Kb/s	75	1200
	110	1800
	134.5	2400
	150	4800) output only
	300	9600) output only
	600	19200) output only
	All above in bits/second	

The data format required of all devices is bit serial; each character indicates its own beginning by means of a start bit preceding the data and includes one or more stop bits at the end of the character.

Given these characteristics, then, the controller will connect to the great majority of normal terminal devices such as Teletypes, alphanumeric CRT units, and modems, and also (with suitable remote interface units) to many peripheral devices such as card readers, line printers, and graphics terminals. Either full or half duplex devices can be accommodated. The standard TIP program cannot deal with a magnetic tape unit through the MLC. However, as a special

option, and with the use of additional core memory, standard Honeywell tape drives can be connected to the TIP as normal peripherals.

The individual terminal line levels are consistent with EIA RS-232C convention. Data rates and character length are individually set for each line *by the program*. For incoming asynchronous lines, the program includes the capability for detecting character length and line data rate as discussed below.

Logically, the controller consists of 64 input ports and 64 output ports. Each input/output pair is brought out to a single connector which is normally connected to a single device. However, by using a special "Y" cable, the ports may go to completely separate devices of entirely different properties. Thus, *input* port 16 may connect to a slow, asynchronous, 5-bit character keyboard while *output* port 16 connects to a high speed, synchronous display of some sort. In order to achieve this flexibility, the MLC stores information about each input and each output port and the program sets up this information for each half of each port in turn as it turns the ports "ON."

Several aspects of the MLC design are noteworthy. The central logic treats each of the 64 ports in succession, each port getting 800 ns of attention per cycle. The port then waits the remainder of the cycle (51.2 μs) for its next turn. For both input and output, two full characters are buffered in the central logic, the one currently being assembled or disassembled and one further character to allow for delays in memory accessing.

During input, characters from the various lines stream into a tumble table in memory on a first come, first served basis. Periodically a clock interrupt causes the program to switch tables and look for input.

Output characters are fed to all lines from a single output table. Ordering the characters in this table in such a way as to keep a set of lines of widely diverse speeds solidly occupied is a difficult task. To assist the program in this, a novel mechanism has been built into the MLC hardware whereby each line, as it uses up a character from the output table, enters a request consisting of its line number into a "request" table in memory. This table is periodically inspected by the program and the requests are used in building the next output table with the characters in proper line sequence.

The design of the terminal interface portion of the MLC is modular. Each Line Interface Unit (LIU) contains all the logic required for full duplex, bit serial communication and consists of a basic bi-directional data section and a control and status section. The data section contains transmit and receive portions each with clock and data lines. For asynchronous devices the clock line is ignored and timing is provided by the MLC itself. (For received asynchronous characters, timing is triggered by the leading edge of the start bit of each character.)

The control and status monitor functions are provided for modems as required by the RS-232C specification. Four outputs are available for control functions and six inputs are available to monitor status. The outputs are under program control and are available for non-standard functions if the data terminal is not a modem. For example, these lines could be used to operate a local line printer. RS-232C connectors are mounted directly on the LIU cards. To allow for variations in terminal and modem pin assignments, the signals are brought to connector pins via jumpers on the card.

2. SOFTWARE SUMMARY

Because the terminals connected to a TIP communicate with Hosts at remote sites, the TIP, in addition to performing the IMP function, also acts as intermediary between the terminal and the distant Host. This means that network standards for format and protocol must be implemented in the TIP. One can thus think of the TIP software as containing both a very simpleminded mini-Host and a regular IMP program.

Figure 2-1 gives a simplified diagrammatic view of the program. The lower block marked "IMP" represents the usual IMP program. The two lines into and out of that block are logically equivalent to input and output from a Host. The code conversion blocks are in fact surprisingly complex and include all of the material for dealing with diverse (and perverse) types of terminals.

As the user types on the keyboard, characters go, via input code conversion, to the input block. Information for remote sites is formed into regular network messages and passes through the OR switch to the IMP program for transmittal. Command characters are fed off to the side to the command block where commands are decoded. The commands are then "performed" in that they either set some appropriate parameter or a flag which calls for later action. An example of this is the LOGIN command. Such a command in fact triggers a complex network protocol procedure, the various steps of which are performed by the PROTOCOL block working in conjunction with the remote Host through the IMPs. As part of this process an appropriate special message will be sent to the terminal via the Special Messages block indicating the status (success, failure, etc.) of the procedure.

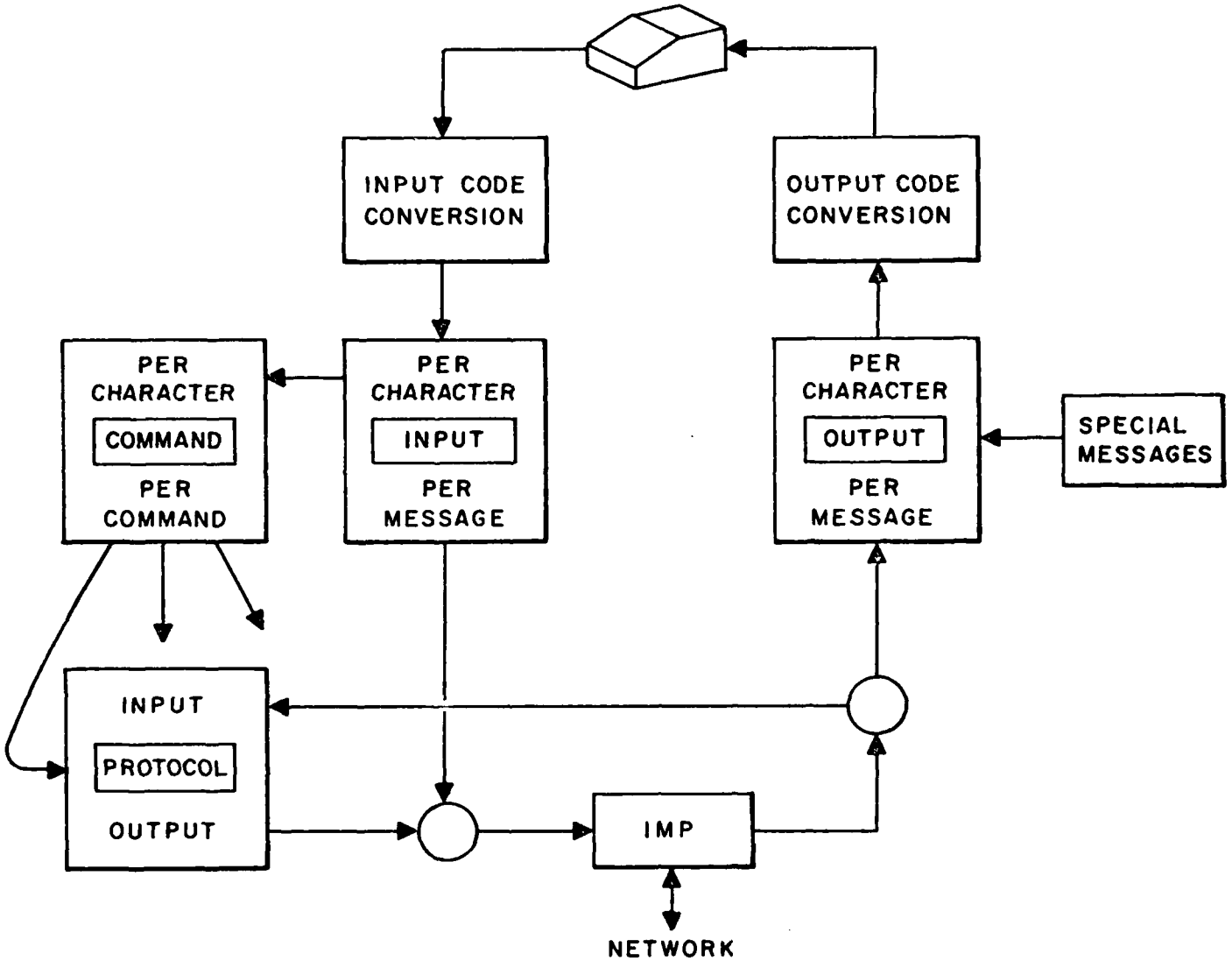


FIGURE 2-1 BLOCK DIAGRAM OF TIP PROGRAM

Once connection to a remote Host is established, regular messages flow directly through the Input block and on through the IMP program. Returning responses come in through the IMP, into the OUTPUT program where they are fed through the OUTPUT Code Conversion block to the terminal itself.

3. PERFORMANCE SUMMARY

The program can handle approximately 100 kilobits of one-way terminal traffic provided the message size is sufficiently large that per-message processing is amortized over many characters. Overhead per message is such that if individual characters are sent one at a time there is a loss of somewhere between a factor of ten and twenty in bandwidth. A different way to look at program performance is to observe that the per-character processing time is about 75 μs.

These figures ignore the fact that the machine must devote some of its bandwidth to acting as an IMP, both for terminal traffic and for regular network traffic. About 5% of the machine is lost to acting as an IMP, even in the absence of traffic. If there is network traffic, more of the machine bandwidth is used up. Five hundred kilobits of two-way phone line and Host traffic saturates the machine without any terminal traffic*.

In addition to bandwidth which goes into the IMP part of the job, another 10 percent of the (total) machine is taken up simply in fielding clock interrupts from the Multi-Line Controller. This again is bandwidth used in idling even with no actual terminal traffic.

The following formula summarizes, approximately, the bandwidth capabilities:

$$P + H + 11T \leq 850$$

*The number 500 kilobits is for full size (8000 bit) messages. Shorter messages use up more capability per bit and thus reduce the overall bandwidth capability.

where:

P = total phone line traffic (in kilobits/sec) wherein, for example, a full duplex 50 Kb phone line counts as 100;

H = total Host traffic (in kilobits/sec) wherein the usual full duplex Host interface, with its usual 10 μs/bit setting, counts as 200; and

T = total terminal traffic (in kilobits/sec) wherein an ASCII terminal such as a (110-baud) full-duplex Teletype (ASR-33) counts as twice its baud rate (i.e., 0.220 Kb).

This means that it takes eleven times as much program time to service every terminal character as it does to service a character's worth of phone line or Host message.

A further factor that influences terminal traffic handling capability has to do with the terminals themselves. Certain types of terminals require more attention from the program than others, independent of their speed but based rather on their complexity. In particular, for example, while an IBM 2741 nominally runs at 134.5 bits per second, the complexity is such that it uses nearly three times the program bandwidth that would be used in servicing a half-duplex ASCII terminal of equivalent speed. Allowances for such variations must be made in computing the machine's ability to service a particular configuration. It must be borne in mind that all of these performance figures are approximations and that the actual rules are extremely complex.

4. SUMMARY OF PROTOCOL DESIGN DECISIONS AND PROTOCOL DEVIATIONS

This section discusses the Terminal IMP's implementation of its Network Control Program (NCP) for the Host/Host Protocol [1], Initial Connection Protocol (ICP) [2], and TELNET [3]. Included are descriptions of the design decisions made where such decisions are permitted by the protocols, and of instances of non-compliance with the protocols.

Most of the choices made during protocol implementation on the Terminal IMP were influenced strongly by storage limitations. The Terminal IMP has no bulk storage for buffering, and has only 12 kilowords of 16-bit words available for both device I/O buffers and program. The program must drive up to 63 terminals which will generally include a variety of terminal types with differing code sets and communication protocols (e.g., the IBM 2741 terminals). In addition, the Terminal IMP must include a rudimentary language processor which allows a terminal user to specify parameters affecting his network connections. Since the Terminal IMP exists only to provide access to the network for 63 terminals, it must be prepared to maintain 126 (simplex) network connections at any time; thus each word stored in the NCP tables on a per-connection basis consumes a significant portion of the Terminal IMP memory.

It should be remembered that the Terminal IMP is designed to provide access to the network for its users, <u>not</u> to provide service to the rest of the network. Thus the Terminal IMP does not contain programs to perform the "server" portion of the ICP; in fact, it does not have a "logger" socket.

4.1 Design Decisions

4.1.1. The Terminal IMP ignores incoming ERR commands and does not output ERR commands.

4.1.2. The Terminal IMP assumes that incoming messages have the format and contents demanded by the relevant protocols. For example, the byte size of incoming TELNET messages is assumed to be 8. The major checks which the Terminal IMP does make are:

1) If an incoming control message has a byte count greater than 120 then it is discarded.

2) If a control command opcode greater than 13 is found during the processing of a control message then the remainder of the control message is discarded.

3) If an incoming data message has a byte count indicating that the bit allocation for the connection is exceeded (based on the assumed byte size) then the message is discarded.

4.1.3 If one control message contains several RST commands only one RRP is transmitted. If several control messages, each containing RST commands, arrive "close together" only one RRP is returned. [The actual implementation is to set a bit each time a RST is found (in "foreground") and to reset the bit when a RRP is sent (in "background")].

4.1.4 Socket numbers and the link for receive connections are pre-assigned based on the hardware "physical address" (in the terminal multiplexing device) of the terminal. The high order 16 bits of the socket number give the device number (in the range 0-63) and the low order 16 bits are normally 2 or 3 depending on the socket's gender (zero is also used during ICP); the TIP requires servers to send data to it on link "device number" +2 [range 2-65].

4.1.5 During ICP, with the Terminal IMP as the user site, the Terminal IMP follows the "Listen" option rather than the "Init" option (as described at the top of page 3 of [2]). In other words, the Terminal IMP does not issue the RFCs involving sockets U+2 and U+3 except in response to incoming RFCs involving those sockets.

4.1.6 The TIP clears out its connection tables as it sends an RST, and then immediately continues operations with the affected Host. RRP's are ignored.

4.2 Deviations from Protocols

4.2.1 The Terminal IMP does not guarantee to issue CLS commands in response to "unsolicited" RFCs. There are currently several ways to "solicit" an RFC, as follows:

1) A terminal user can tell the Terminal IMP to perform the ICP to the TELNET Logger at some foreign Host. This action "solicits" the RFCs defined by the ICP.

2) A terminal user can send an RFC to any particular Host and socket he chooses. This "solicits" a matching RFC.

3) A terminal user can set his own receive socket "wild." This action "solicits" an STR from anyone to his socket. Similarly, the user can set his send socket "wild" to "solicit" an RTS.

If the terminal IMP receives a "solicited" RFC, it handles it in accordance with the protocol. For unsolicited RFCs, the Terminal IMP maintains a two-entry reply queue. When an unsolicited RFC is received, permanent information is placed on the queue if there is room; if not, the RFC is ignored. In the "background", the queue is emptied by constructing the appropriate CLSs.

4.2.2. After issuing a CLS for a connection, the Terminal IMP will not wait forever for a matching CLS. There are two cases:

1) The Terminal IMP has sent an RFC, grown tired of waiting for a matching RFC, and therefore issued a CLS.

2) The Terminal IMP has sent a CLS for an established connection (matching RFCs exchanged).

In either of these cases the Terminal IMP will wait for a matching CLS for a "reasonable" time (probably 30 seconds to one minute) and will then "forget" the connection. After the connection is forgotten, the Terminal IMP will consider both sockets involved to be free for other use.

Because of program size and table size restrictions, the Terminal IMP assigns socket numbers to a terminal as a direct function of the physical address of the terminal. Thus (given this socket assignment scheme) the failure of some foreign Host to answer a CLS could permanently "hang" a terminal. It might be argued that the Terminal IMP could issue a RST to the offending Host, but this would also break the connections of other terminal users who might be performing useful work with that Host.

4.2.3 The Terminal IMP ignores all RET commands. The Terminal IMP cannot buffer very much input from the network to a given terminal because of core size limitations. Accordingly, the Terminal IMP allocates only one message and a small number of bits on each connection for which the Terminal IMP is the receiver. The Terminal IMP attempts to keep the usable bandwidth as high as possible by sending a new allocation, which brings the total allocation up to the maximum amount, each time that:

1) one of the two buffers assigned to the terminal is empty, and

2) the allocations are below the maxima.

Thus, if a spontaneous RET were received, the reasonable thing for the Terminal IMP to do would be to immediately issue a new ALL. However, if a foreign Host had some reason for issuing a first spontaneous RET, it would probably issue a second RET as soon as it received the ALL. This would be likely to lead to an infinite (and very rapid) RET-ALL loop between the two machines, chewing up a considerable portion of the Terminal IMP's bandwidth. Since the Terminal IMP can't "afford" to communicate with such a Host, it ignores all RETs.

4.2.4. The Terminal IMP ignores all GVB commands. Implementation of GVB appears to require an unreasonable number of instructions and, at the moment at least, no Host appears to use the GVB command. If we were to implement GVB we would always RET all of both allocations and this doesn't seem very useful.

4.2.5. The Terminal IMP does not handle a total bit-allocation greater than 65,534 (2^{16}-2) correctly. If the bit-allocation is ever raised above 65,534 the Terminal IMP will treat the allocation as infinite. This treatment allows the Terminal IMP to store the bit allocation for each connection in a single word, and to avoid double precision addition and subtraction. Our reasons for this decision are:

1) We save more than 100 words of memory which would be required for allocation tables and for double precision addition/subtraction routines.

2) Our experience indicates that very few Hosts (probably one at most) ever raise their total bit allocation above 65,534 bits.

3) We expect that any Host that ever raises its bit allocation above 65,534 would probably be willing to issue an infinite bit allocation if one were provided by the protocol. Once the bit allocation is greater than about 16,000, the message allocation (which the Terminal IMP handles correctly) is a more powerful method of controlling network loading of a Host system than bit allocation.

4.2.6. The Terminal IMP ignores the "32-bit number" in the ICP. When the Terminal IMP (the "user site") initiates the Initial Connection Protocol the actual procedure is to send the required RTS to the logger socket of the user-specified foreign Host and simultaneously to set the terminal user's send and receive sockets in a state where each will accept any RFC from the specified Host. The 32-bit socket number transmitted over the logger connection is ignored, and the first RTS and STR addressing the user's sockets will be accepted (and answered with matching RFCs).

The ICP allows the foreign Host to transmit the RFCs involving Terminal IMP sockets "U+2" and "U+3" at any time after receipt of the RFC to the (foreign Host's) logger socket. In particular, the RFCs may arrive at the Terminal IMP before the 32-bit number. In the case of a "normal" foreign Host, the first incoming RFCs for sockets U+2 and U+3 will come from the sockets indicated by the 32-bit number, so it doesn't matter if the number is ignored. In the case of a pathologic foreign Host, a potentially infinite number of "wrong" RFCs involving U+2 and U+3 may arrive at the Terminal IMP before the 32-bit number is sent. The Terminal IMP would be required to store this stream of RFCs pending arrival of the 32-bit number, then issue CLS commands for all "wrong" RFCs. However, the Terminal IMP does not have infinite storage available for this purpose (it is also doubtful that a terminal user really wants to converse with a pathologic foreign Host) so the Terminal IMP assumes that the foreign Host is "normal" and ignores the 32-bit number.

4.2.7 The terminal IMP does not guarantee correct response to ECO commands. ECOs are handled in exactly the same fashion as RSTs [a bit which is set is foreground and cleared in background], but for ECO, this approach constitutes a violation.

The reaons for this method of implementation is that to guarantee correct response to ECO in all cases requires an infinite amount of storage. For example, suppose Host A sends control messages, each containing an ECO command, to Host B at the rate of one per second, but that Host A accepts messages from the network as slowly as possible (one every 39 seconds, say). Then Host B has only three choices which do not violate protocol:

1) Declare itself dead to the network (i.e., turn off its Ready line), thereby denying all its users use of the network.

2) Refuse to accept messages form the network faster than the slowest possible foreign Host (i.e., about one every 39 seconds). If Host B is a Terminal IMP, this is almost certainly slow enough to soon reach a steady state of no users.

3) Implement "infinite" storage for buffering messages.

Since it is clear that none of the "legal" solutions are possible, we have decided to provide buffering for one ERP to each possible Host, which should be adequate well over 99% of the time.

4.2.8 INSs and DMs are counted only mod4 with the counter nominally at Ø and INSs decrementing it and DMs incrementing it. When the counter is at 2 or 3, output is suppressed. Thus, if there are a large number of TELNET SYNCHs in progress (to a single port) simultaneously, some of the SYNCHs will not have been handled correctly.

The reason for this method of implementation is that protocol requires a potentially infinite counter to keep track of INSs that have not yet been matched. Further, since this method will always fall back in synch when the matching DMs eventually do come in, the only effect of this is occasionally not suppressing output when it should have been.

5. References and TIP Bibliography

[1] NIC 8246, Host/Host Protocol for the ARPA Network, January 1972.

[2] NIC 7101, Official Initial Connection Protocol, June 1971.

[3] NIC 9348, Ad Hoc Telnet Protocol, April 1972.

[4] The BBN TIP Hardware Manual, BBN Report No. 2184.

[5] Specifications for the Interconnection of Terminals and the Terminal IMP, BBN Report No. 2277.

[6] Terminal Interface Message Processor User's Guide, BBN Report No. 2183.

[7] S.M. Ornstein et al, The Terminal IMP for the ARPA Computer Network, Proc. AQIPS 1972 SJCC., pp. 243-254.

[8] N.W. Mimno et al, Terminal Access to the ARPA Network: Experience and Improvements, Proc. Seventh Annual IEEE Computer Society International Conference, pp. 39-43.

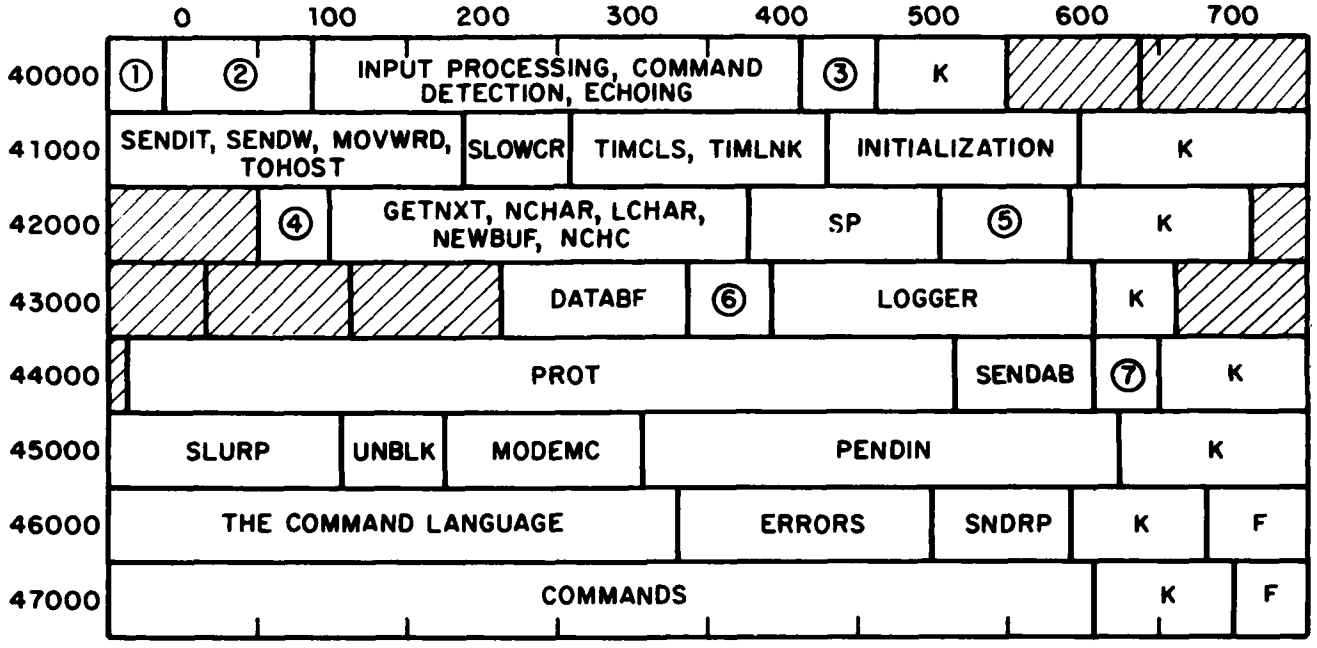
[9] R.E. Kahn, Terminal Access to the ARPA Computer Network, Computer Networks, Courant Computer Symposium 3, Courant Institute, New York.

[10] TIP User's Group Notes, Available from the Network Information Center, Stanford Research Institute, Menlo Park, California 94025.

6. Storage Layout

There follow three figures which illustrate the layout of core storage for the TIP software system.

The numbers down the left side and across the top of the tables are given in octal and represent memory addresses.



K = CONSTANTS

F = FREE

① BACKGROUND LOOP

② CLOCK INTERRUPT, MLC INPUT, CLOCK "LOOP", CLOCK EXIT

③ MLC OUTPUT

④ MLC OUTIN

⑤ NETCOM, ECHOING

⑥ OUNEW

⑦ FIRE & TILEAD

HOST AND DEVICE STATUS TABLE

FIGURE 6-1(A) STORAGE LAYOUT

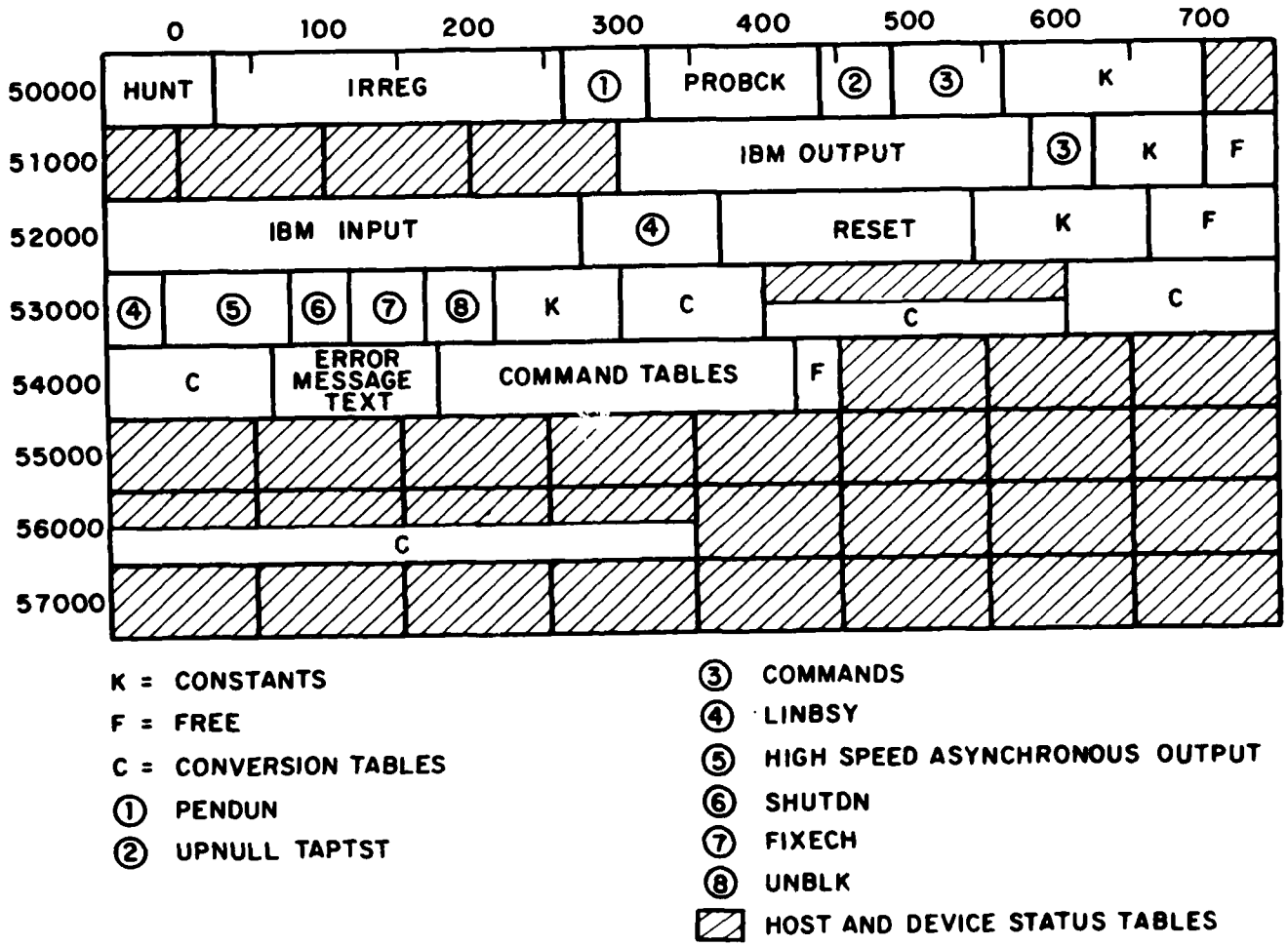


FIGURE 6 - 1(B) STORAGE LAYOUT

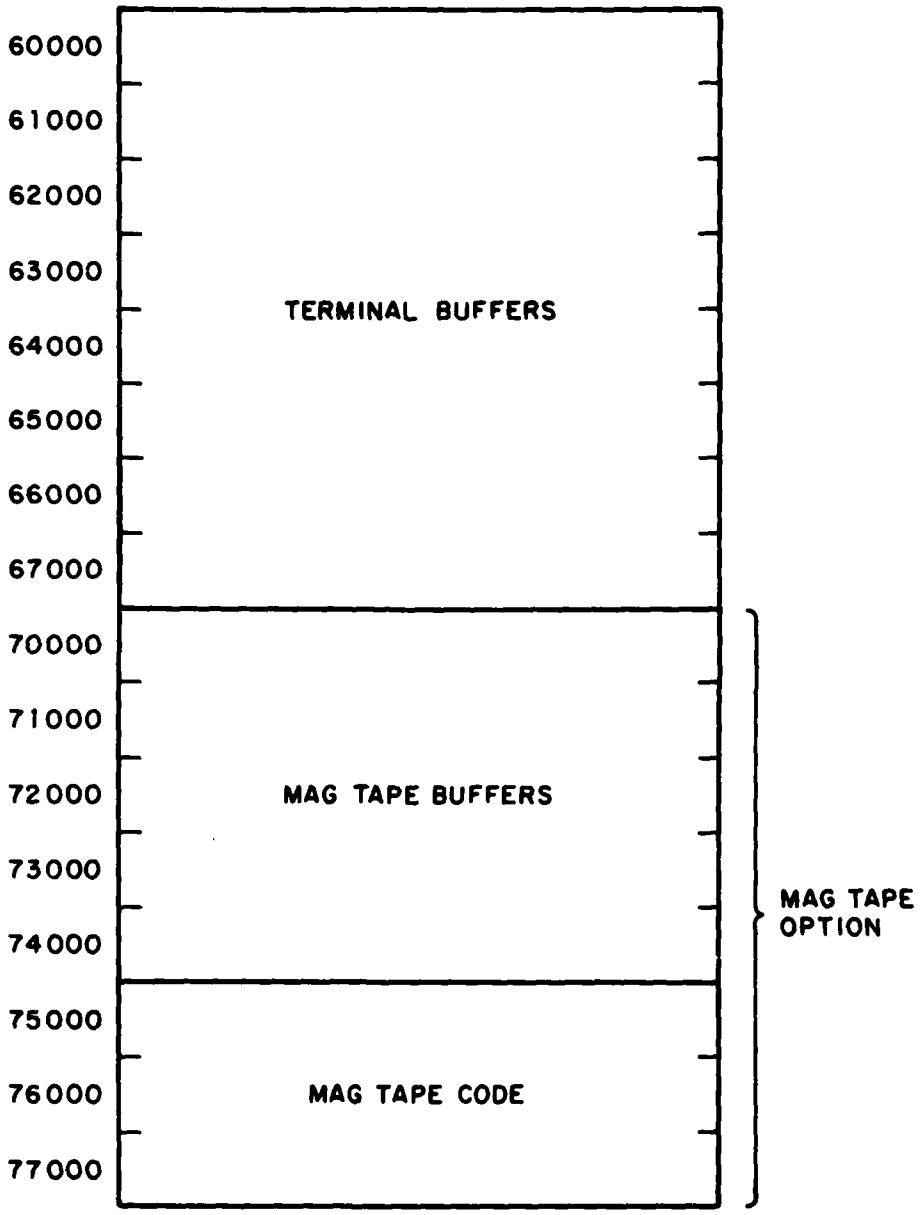


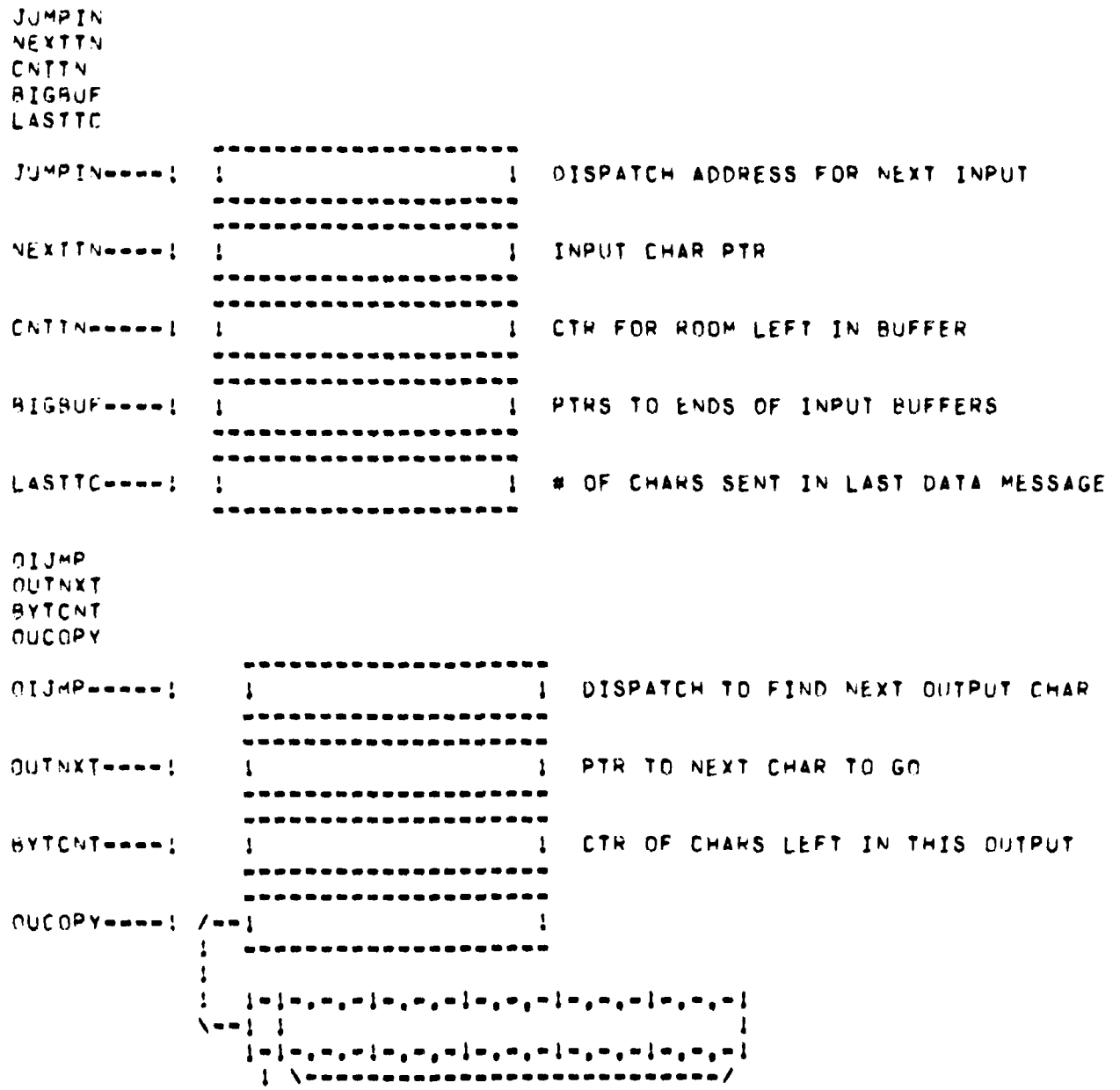
FIGURE 6-1(C) STORAGE LAYOUT

7. Data Structures

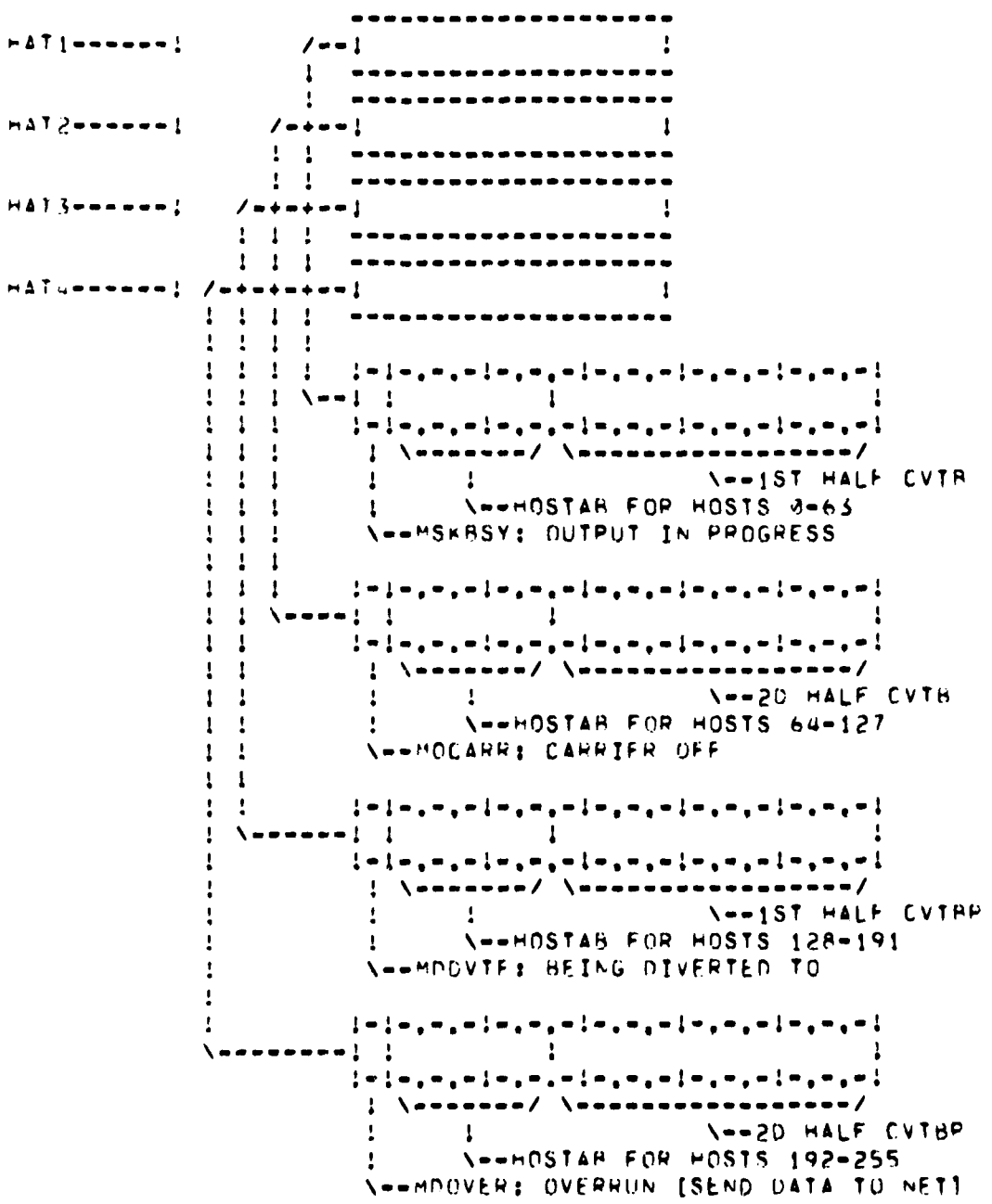
Following are the formats of the individual entries of every important table or data structure in the TIP software system. These charts are useful in studying the TIP system listing and the detailed software descriptions.

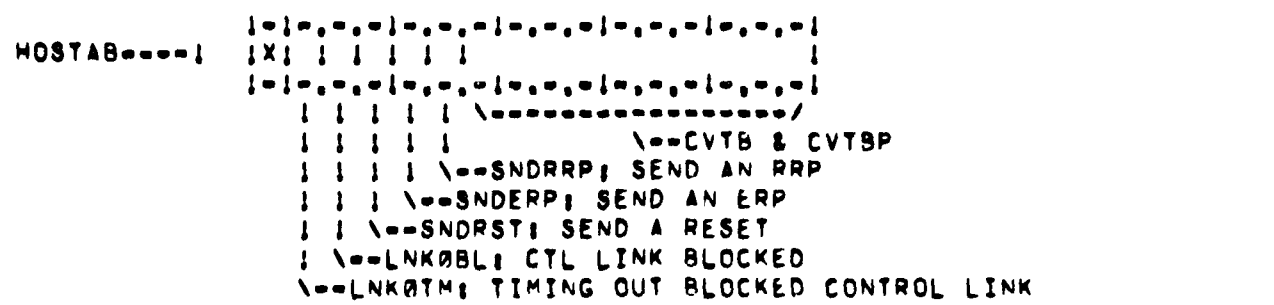
The tables are invariably sixty-four entries long or, as is occasionally necessary, made up of two or more contiguous tables sixty-four entries long. Often several tables only a few bits wide are packed in one word.

```
RATE
CODE
SIZE
RHITS
                !-!-.-.-!-.-.-!-.-.-!-.-.-!-.-.-!
DEVICE----!     ! ! ! !       !   !       ! !   !
                !-!-.-.-!-.-.-!-.-.-!-.-.-!-.-.-!
                 ! ! ! \-----/ \-/ \-----/ ! \-/
                 ! ! !     !     !     !     !  \--MRCODE: CODE CONVERSION
                 ! ! !     !     !     !     \--RESERVED FOR MRCODE EXPANSIO
                 ! ! !     !     !     !           N
                 ! ! !     !     !     \--DEVICE INPUT RATE
                 ! ! !     !     \--CHAR SIZE
                 ! ! !     \--MRORAT: DEVICE OUTPUT RATE
                 ! ! \--MDLOG: BEGIN LOGIN
                 ! \--MCOMSP: LOOKING FOR A SPACE IN A COMMAND
                 \--MDFIND: HUNT WHEN DEVICE DISCONNECTS

ECHRD1
CHITS
                !-!-.-.-!-.-.-!-.-.-!-.-.-!-.-.-!
DEVICE----!     !                   !XXX! ! ! ! !
                !-!-.-.-!-.-.-!-.-.-!-.-.-!-.-.-!
                 \----------------/     ! ! ! !
                          !             ! ! ! \--MDADDL: ADD LF'S AFTER C
                          !             ! ! !        R'S
                          !             ! ! \--MDRILD: DEVICE IS RILD
                          !             ! \--MDTOF: TERMINATE ON EOM
                          !             \--MDTOL: TERMINATE ON LINEFEED
                          \--1ST CHAR TO ECHO WITH

ECHRD2
DHITS
                !-!-.-.-!-.-.-!-.-.-!-.-.-!-.-.-!
DEVICE----!     !                   !X! ! ! ! ! !
                !-!-.-.-!-.-.-!-.-.-!-.-.-!-.-.-!
                 \----------------/     ! ! ! ! !
                          !             ! ! ! ! \--MDDRI: DON'T RESET DRUM
                          !             ! ! ! !       INPUT
                          !             ! ! ! \--MDDRUM: DON'T RESET DRUM O
                          !             ! ! !       UTPUT
                          !             ! ! \--MDECH1: VIRTUAL FULL DUPLEX
                          !             ! \--MDECH2: PHYSICAL HALF DUPLEX
                          !             \--MDBINO: BINARY OUTPUT MODE
                          \--FINAL CHAR TO ECHO WITH
```

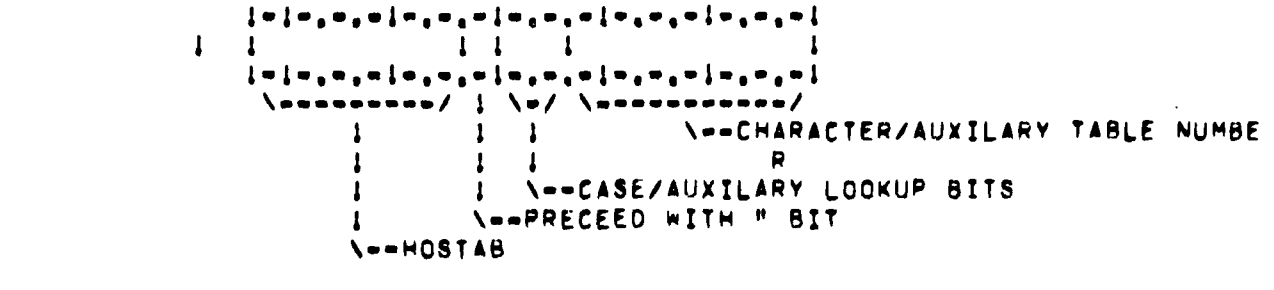

JUMPIN
NEXTTN
CNTTN
BIGBUF
LASTTC
JUMPIN
DISPATCH ADDRESS FOR NEXT INPUT
NEXTTN
INPUT CHAR PTR
CNTTN
CTR FOR ROOM LEFT IN BUFFER
BIGBUF
PTRS TO ENDS OF INPUT BUFFERS
LASTTC
OF CHARS SENT IN LAST DATA MESSAGE
OIJMP
OUTNXT
BYTCNT
OUCOPY
OIJMP
DISPATCH TO FIND NEXT OUTPUT CHAR
OUTNXT
PTR TO NEXT CHAR TO GO
BYTCNT
CTR OF CHARS LEFT IN THIS OUTPUT
OUCOPY
PTRS TO ENDS OF OUTPUT BUFFERS
TBI WHICH DOUBLE BUFFER IS IN USE

```
HOSTAB
CVTB
CVTBP
HAT1
HAT2
HAT3
HAT4
```

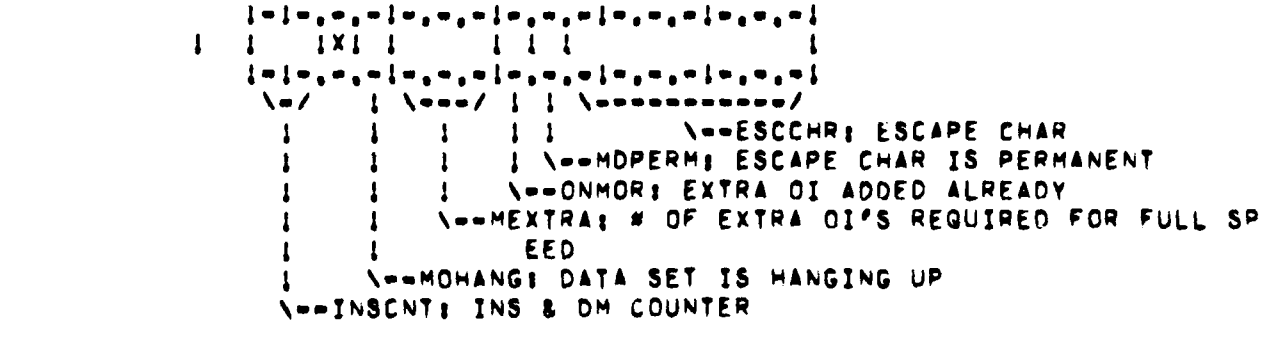


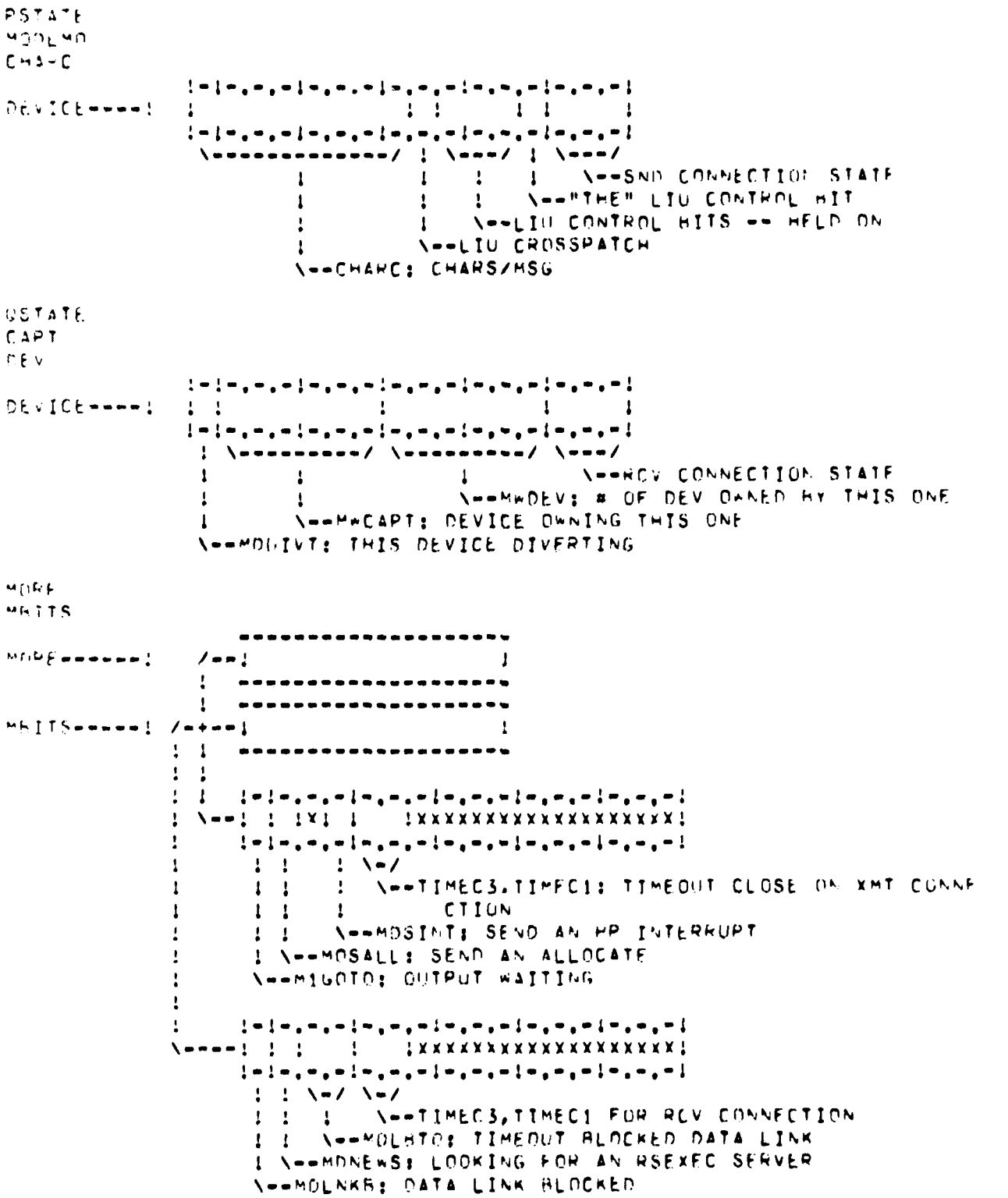


CVTB
CVTBP



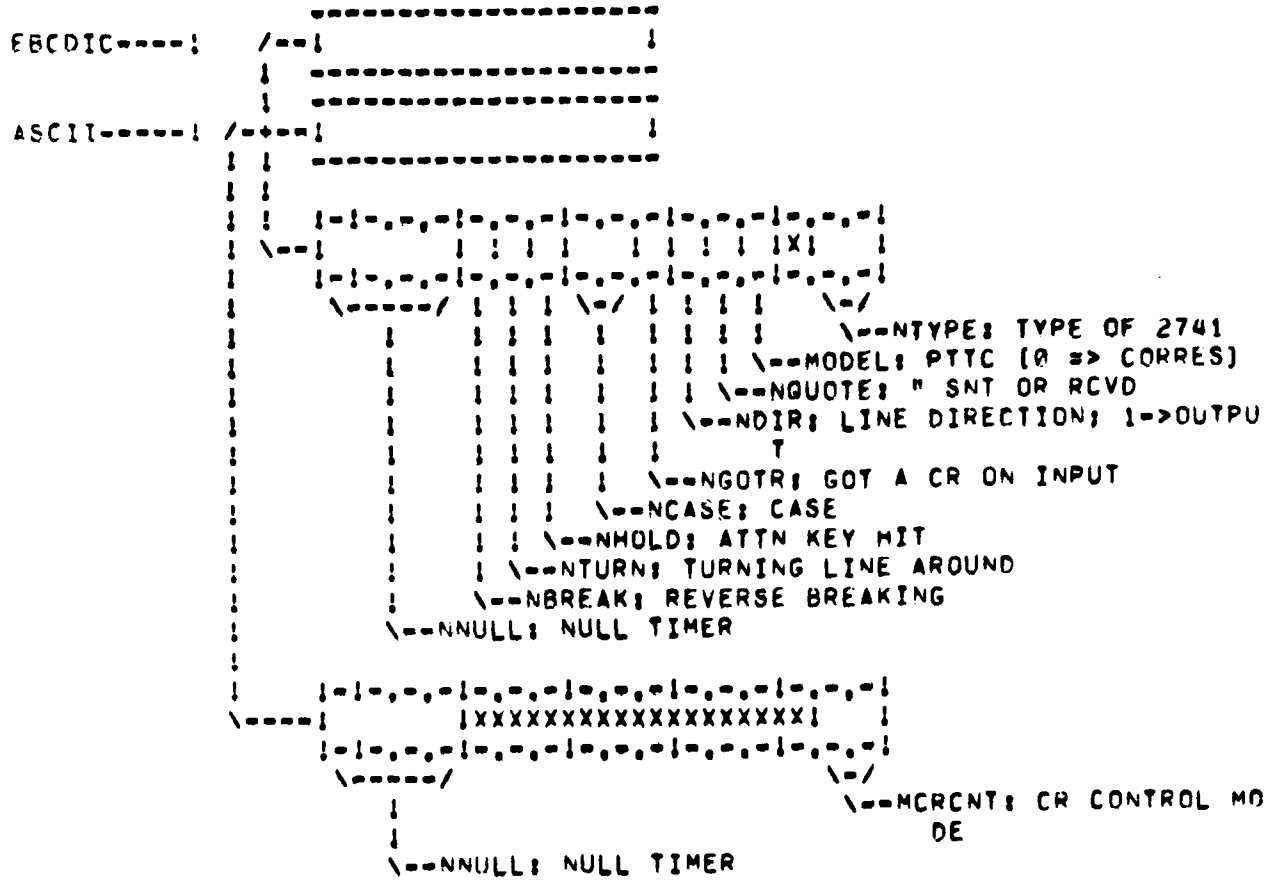
DEVTBS



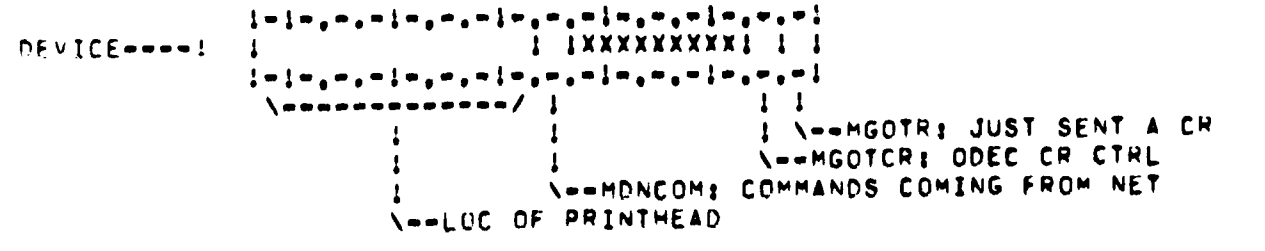

PSTATE
MODEMD
CHS-C
DEVICE----!
USTATE
CAPT
DEV
DEVICE----!
MORE
MBITS
MORE------!
MBITS-----!
MDSINT: SEND AN HP INTERRUPT
MDSALL: SEND AN ALLOCATE
M1GOTO: OUTPUT WAITING
MDLNKB: DATA LINK BLOCKED

```
SOCKS1
SOCKS2
SOCKR1
SOCKR2
                 ------------------
SOCKS1----!      !                !  SEND SOCKET, WORD 1
                 ------------------
                 ------------------
SOCKR1----!      !                !  RECEIVE SOCKET, WORD 1
                 ------------------
                 ------------------
SOCKS2----!      !                !  SEND SOCKET, WORD 2
                 ------------------
                 ------------------
SOCKR2----!      !                !  RECEIVE SOCKET, WORD 2
                 ------------------

HOSTS
LINK
                 !-!-.-.-!-.-.-!-.-.-!-.-.-!-.-.-!-.-.-!
DEVICE----!      !                !                !
                 !-!-.-.-!-.-.-!-.-.-!-.-.-!-.-.-!-.-.-!
                  \------------/ \------------/
                         !                \--HOST TO SEND TO
                         \--LINK TO SEND ON

HOSTR
COMAND
                 !-!-.-.-!-.-.-!-.-.-!-.-.-!-.-.-!-.-.-!
DEVICE----!      !                !                !
                 !-!-.-.-!-.-.-!-.-.-!-.-.-!-.-.-!-.-.-!
                  \------------/ \------------/
                         !                \--HOST TO RCV FROM
                         \--INDEX FOR COMMAND DECODE

ALLOC
ALLOCM
ALLOCO
                   ------------------
ALLOC-----!        !                !  BIT ALLOC LEFT (-1=INF)
                   ------------------
                   ------------------
ALLOCM----!        !                !  MSG ALLOC LEFT (-1=INF)
                   ------------------
                   ------------------
ALLOCO----!    /--!                !
               !   ------------------
               !
               !   !-!-.-.-!-.-.-!-.-.-!-.-.-!-.-.-!-.-.-!
               \--!   !                                  !
                  !-!-.-.-!-.-.-!-.-.-!-.-.-!-.-.-!-.-.-!
                   ! \--------------------------/
                   !                \--# OF CHARS IN LAST MSG RCVD
                   \--ALLOCATE SENT IN LAST CONTROL MSG
```

NN


EBCDIC
ASCII
NTYPE: TYPE OF 2741
MODEL: PTTC (0 => CORRES)
NQUOTE: " SNT OR RCVD
NDIR: LINE DIRECTION; 1=>OUTPU
NGOTR: GOT A CR ON INPUT
NCASE: CASE
NHOLD: ATTN KEY HIT
NTURN: TURNING LINE AROUND
NBREAK: REVERSE BREAKING
NNULL: NULL TIMER
MCRCNT: CR CONTROL MO
DE
NNULL: NULL TIMER

M


DEVICE
MGOTR: JUST SENT A CR
MGOTCR: ODEC CR CTRL
MDNCOM: COMMANDS COMING FROM NET
LOC OF PRINTHEAD

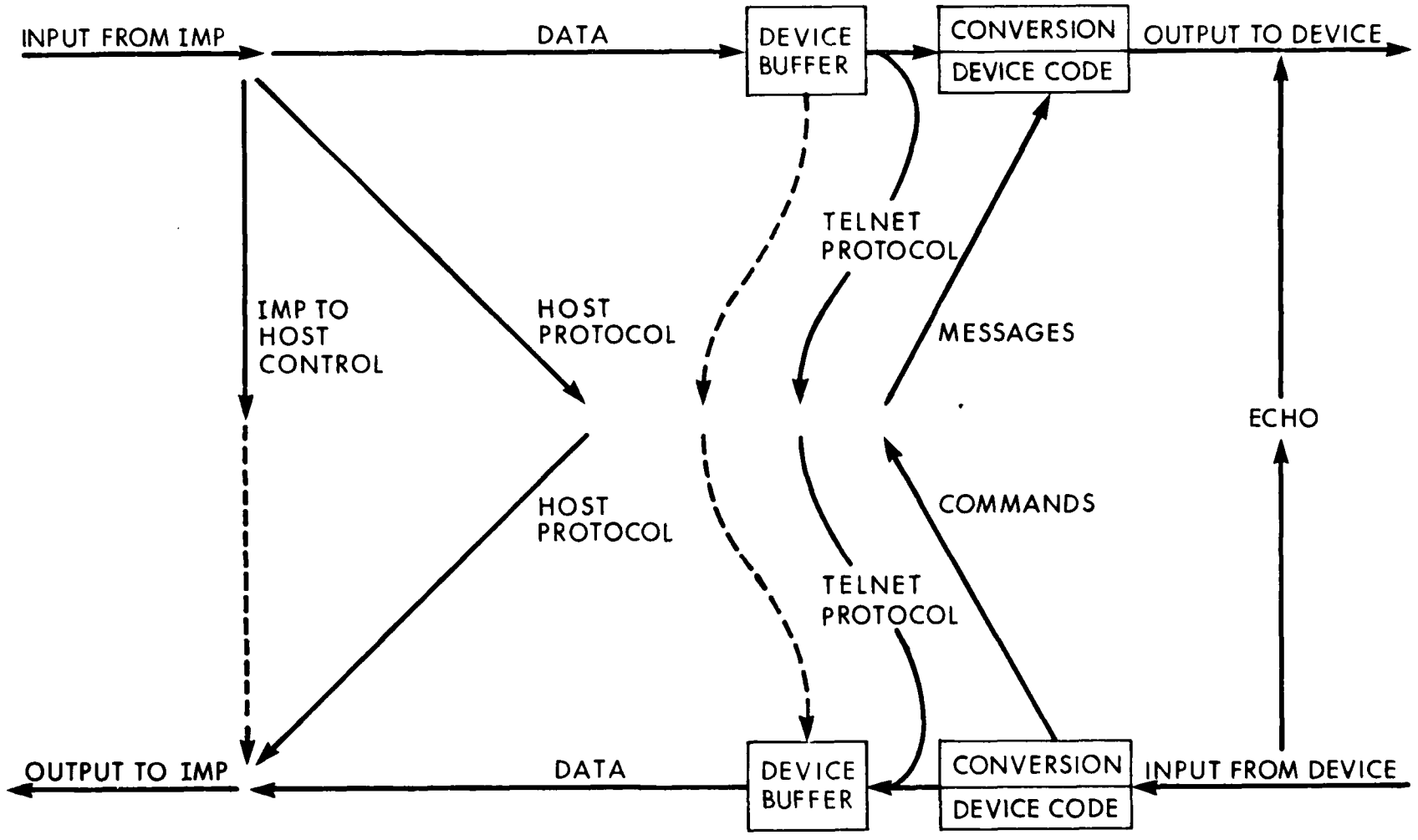
8. Detailed Software Description

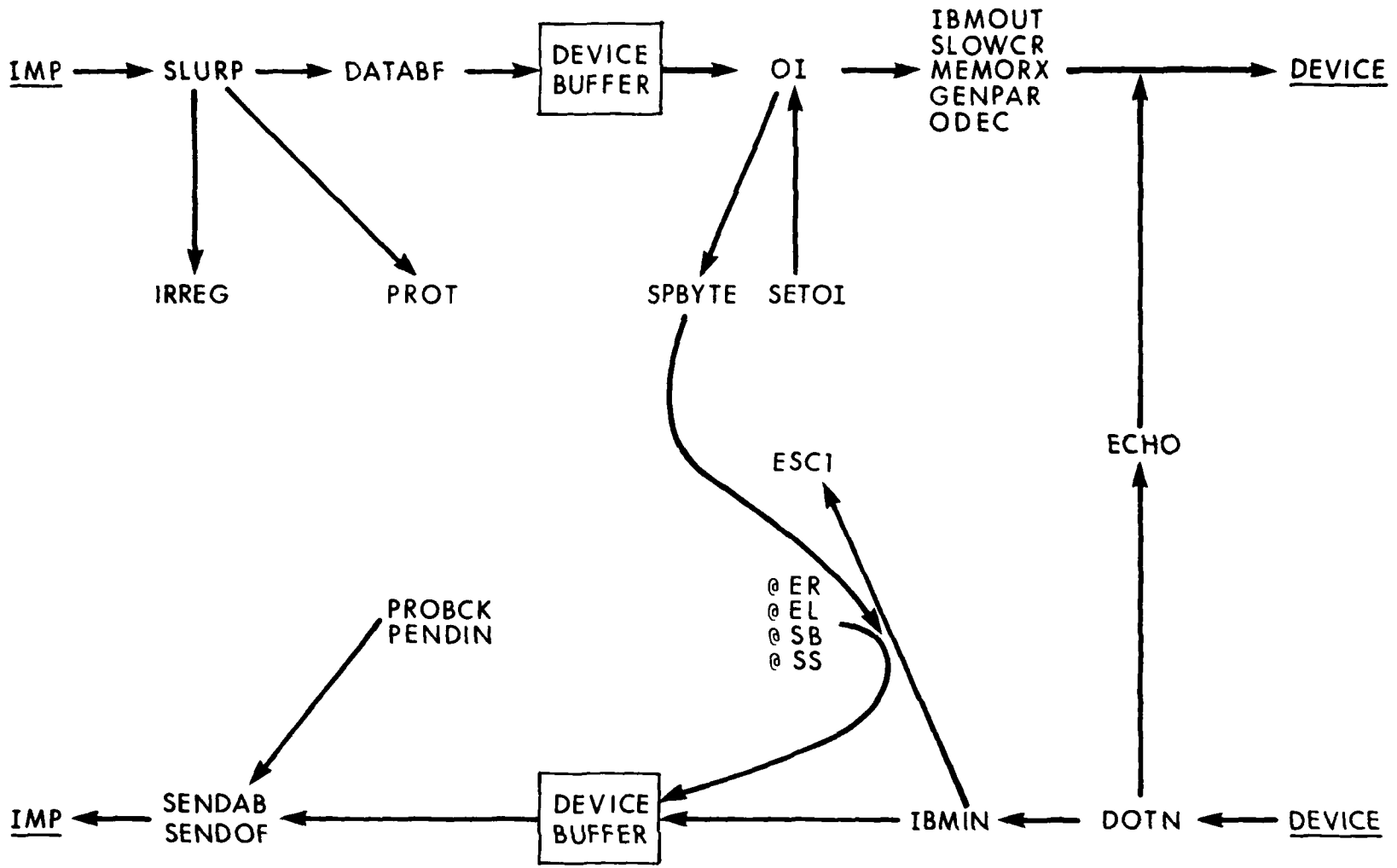
Section 8.1 gives an overview of the various functional modules in the TIP program and the main routines which perform these functions. Section 8.2 contains the bulk of the material in Section 8, the detailed software descriptions themselves. Section 8.3 is an index of the "labels" used in Section 8.2.

8.1 Outline of Program's Functional Structure

The figure gives a broad outline of the data paths in the IMP system software. Data paths are marked with directed arcs. A few important paths (marked with dashed arcs) which control the flow over the paths are also shown.

The figure after next indicates some of the routines along the data paths and serves as an overview of the detailed descriptions of the next section. The following two figures should be studied together.


INPUT FROM IMP
DATA
DEVICE BUFFER
CONVERSION
DEVICE CODE
OUTPUT TO DEVICE
TELNET PROTOCOL
IMP TO HOST CONTROL
HOST PROTOCOL
MESSAGES
ECHO
HOST PROTOCOL
COMMANDS
TELNET PROTOCOL
OUTPUT TO IMP
DATA
DEVICE BUFFER
CONVERSION
DEVICE CODE
INPUT FROM DEVICE


IMP
SLURP
DATABF
DEVICE BUFFER
OI
IBMOUT
SLOWCR
MEMORX
GENPAR
ODEC
DEVICE
IRREG
PROT
SPBYTE
SETOI
ECHO
ESC1
@ ER
@ EL
@ SB
@ SS
PROBCK
PENDIN
IMP
SENDAB
SENDOF
DEVICE BUFFER
IBMIN
DOTN
DEVICE

8.2 Detailed Descriptions

The detailed descriptions in the rest of this section are a mixture of descriptive prose and diagrams. Very few of the diagrams are precise flow charts of the actual code; instead, the diagrams vary from functional descriptions of what the routine under consideration does, to state diagrams for the conceptual finite state machines that the routine implements, to logic diagrams for the routine. Study of the remainder of this section assumes concurrent study of the program listing and the formats of the program data structures. In the following diagrams, wherever reasonable, the diagram labels are keyed to the program labels found in the listing for convenient cross-referencing.

Many of the diagrams are given in an ALGOL-like notation where labels appear in the left column and "goto"s and "if"s are used to indicate branches in program flow. This notation is used instead of a more conventional two-dimentional notation to facilitate computer maintenance of the diagrams. In the absence of "goto"s, "call"s, etc. program flow is assumed to go down the page and from one page to the next.

The program descriptions in the rest of this section are generally organized by program priority level (initialization, MLC clock interrupt, background, MLC output interrupt, and mag tape option) as shown in the following hierarchical outline. Again, this organization is to facilitate cross-referencing the descriptions with the listing.

8.2.1 Initialization
8.2.2 Clock Level

8.2.2.1 MLC Input

8.2.2.1.1 Packing data into input buffers
8.2.2.1.2 Echo initiation
8.2.2.1.3 User command processing

8.2.2.1.3.1 Command interpretation
8.2.2.1.3.2 Command execution

8.2.2.1.4 Rate initialization for "hunting" devices
8.2.2.1.5 Input conversion

8.2.2.1.5.1 EBCDIC to ASCII

8.2.2.2 MLC Output (Outin's)

8.2.2.2.1 2741 line control
8.2.2.2.2 Telnet protocol

8.2.2.2.2.1 FIXECH

8.2.2.2.3 Echoes
8.2.2.2.4 Messages to devices
8.2.2.2.5 Output Conversion

8.2.2.2.5.1 ASCII to EBCDIC Conversion Logic
8.2.2.2.5.2 Slow carriage return
8.2.2.2.5.3 Even Parity
8.2.2.2.5.4 Line Printers

8.2.2.2.5.4.1 MEMORX
8.2.2.2.5.4.2 ODEC

8.2.2.3 Background timer

8.2.3 Background

8.2.3.1 Connection and Host functions

8.2.3.1.1 Device related functions

8.2.3.1.1.1 Transfer of data to the IMP
8.2.3.1.1.2 Allocates
8.2.3.1.1.2 News

8.2.3.1.2 Host related functions

8.2.3.1.2.1 Send INS's
8.2.3.1.2.2 Send ERP's, RST's, and RRP's
8.2.3.1.2.3 Send CLS's for unsolicited RFC's and CLS's

8.2.3.2 Connection Control
8.2.3.3 Initial Connection Protocol
8.2.3.4 Send data to IMP

8.2.3.4.1 Link Ø
8.2.3.4.2 Not link Ø

8.2.3.5 Accept data from IMP

8.2.3.5.1 Irregular Messages
8.2.3.5.2 Regular Messages

8.2.3.5.2.1 Link Ø messages

8.2.3.5.2.1.1 Handling RFC's and CLS's
8.2.3.5.2.1.2 Handling ECO's and RST's
8.2.3.5.2.1.3 Handling Allocates
8.2.3.5.2.1.4 Counting INS's

8.2.3.5.2.2 Messages not on link Ø

8.2.3.5.2.2.1 OUNEW

8.2.3.6 Modem and LIU control

8.2.4 MLC Output interrupt
8.2.5 Magnetic tape option

8.3 Index to Detailed Descriptions

8.2.1 Initialization

Initialization comes in two classes: complete TIP initialization (done by the routine GOGGLE), and device initialization (done by the routine RESET).

GOGGLE is called at the top of the TIP background loop if the TIP (re)initialization flag (TIPFLG) is set. RESET is called by GOGGLE as part of complete TIP initialization, and by MODEMS, HUNT, and RC for device initialization, the latter being the RESET command.

GOGGLE shut off TIP interrupts

reset the MLC

set up Page Ø literals

for all Hosts, mark to send RSTs

initialize device stuff including
call to RESET

zero various flags

initialize MLC output buffer pointers

initialize various co-routines

initialize various timers

set up correct priority
interrupt mask for clock level

initialize MLC IN and OUTIN buffer pointers

initialize Mag tape option if it exists

set up interrupt entrances

return from GOGGLE

The subroutine RESET, following, initializes a device. It is called by various routines, including: GOGGLE (initialization), MODEMC, HUNT, and the user RESET command. RESET makes a distinction between hunting and non-hunting devices; the latter keep certain parameters set even through a reset. On entry, the device to be reset is indicated by the index register.

RESET performs the following functions for all devices, hunting or not:

- initializes internal status bits (allocates, blocked links, etc.);
- sets the escape character to the default unless marked as permanent;
- initializes connections (allocates and links) by calling SHUTDN if not open, or if an open connection exists, initiates a close;
- aborts any login, open, or news request in progress;
- initializes buffer pointers and count;
- releases any devices captured by this one, and uncaptures this device if captured;
- sets up the appropriate dispatch for input characters.

In addition, RESET performs the following functions for hunting devices only:

- changes the device rate to the nominal hunting rate;
- resets the "connection" parameters to their default conditions of not wild, non-binary output, transmit on every character (and not specifically on an EOM or EOL), local echo mode, and add a linefeed after a carriage return.

RESET if device is invalid, return from RESET

initialize status bits

if escape character is permanent, reset escape character to @

RESET5 initialize (close) connections

initialize buffer pointers and count

uncapture the device and any other devices it may have captured

if device is hunting device, goto RESETa

set up correct input dispatch

return from RESET

RESETa set the rate to hunting

initialize the connection parameters

set input dispatch to HUNT

return from RESET

8.2.2 MLC CLOCK Level

The MLC clock level routine, CLOCK, is mostly a straight series of calls to other clock level routines and inline code.

CLOCK	save registers and keys and mask
	set up new mask
	if 40 ticks of PEND clock have not gone by, goto CLOCK4
	reset PEND clock
	reset LOG clock
	reset PEND flag
	mark that TIP is ready (for NCC)
	increment data set control timer
CLOCK4	do MLC input
DOTN2	pack data into device input buffers

```
OOPS    call CLKOI

        call BTIME

        if output not busy, call TOUT
        to restart it

        restore registers, mask, and keys

        return from CLOCK
```

8.2.2.1 MLC Input

The inline code following CLOCK4 in CLOCK performs MLC input as shown.

swap buffers

set up MLC input
buffer pointers to
new input buffer

set up pointer
to input buffer to
be emptied

goto DOTN2

8.2.2.1.1 Packing Data into INPUT Buffers

The input routine starting at DOTN is called on an interrupt basis. Characters input to the MLC are stored in a (double-buffered) tumble table where each entry consists of the character and an index to the originating device. Each clock time of 3.3 msec, one of these tumble tables is processed by the input routines, character-by-character. The disposition of input is indicated by the JUMPIN table which contains a dispatch (jump instruction) for each device; specific dispatches are detailed in the following flow charts. Generally they distinguish from each other: data, commands, echoing or not echoing, ASCII input and input requiring code conversion, input considered 8-bit binary, and the special case of the initial character(s) from a hunting device. Clearly, the dispatch changes as a user sets up and then executes a dialogue with a Host system.

DOTN — increment to get next entry from tumble table

DOTN2 — pick up a character and device number from tumble table and save them

if character is a break, goto BREAK

dispatch via table entry for this device to one of CONECO, CONVT, CONEEE, CONESC, IBMEEE, IBMESC, IBMECO, IBMCON, BINECO, BINCON, or HUNT

```
BREAK     if device is hunting, goto HUNT
          if device is a 2741, goto IBBRAK
          convert character to TELNET break character
          goto NOPE
```

```
BINECO      call ECHO
            to echo character

BINCON      goto NOPE


IBMEEE
IBMESC      call IBMIN to convert
            character and do protocol

            goto ESC


IBMECO
IBMCON      call IBMIN to convert character
            and do protocol

            goto REG
```

```
REG     if device has suppressed intercepting
        commands, goto REGa

        if character is the device's escape
        character, goto ESCAPE

REGa    if character is a LF, goto FEED

        if character is an EOM, goto EOM

NOPE    if there is no space in buffer,
        goto ECHBEL

        store the character in buffer

        if there is no space left, call
        SENDIT to mark it to be sent to net

ADDLF   if not in add LF mode,
        goto DOTN

        if character is a CR, change
        character in tumble table to a LF

        goto DOTN2
```

ECHBEL don't echo the character, but
send out a BEL code (upshift
for IBMs) to show character was discarded

goto DOTN

FEED if device not set to
transmit on LF, goto NOPE

goto EOMa

EOM if device not set to transmit
on EDM, goto NOPE

EOMa set up counter to make buffer look full

goto NOPE

```
CONEEE      call ECHO to echo
            character

CONESC      mask character to
            seven bits

ESC         if character is a CR,
            goto ADDLF

            call ESC1 to interpret
            command character

            if return 1 from ESC1, goto DOTN

ESC2        set up table dispatch
            to expect data

            if character is an esc which
            is to be data, goto NOPE

            terminate command by
            sending device CR-LF

            goto DOTN
```

8.2.2.1.2 Echo initiation

The ECHO subroutine performs local echoing for the TIP. It has table storage space allowing two echo characters to be saved for each device, tables ECHWD1 and ECHWD2. Echo characters take priority over regular data for output.

ECHO

if OI dispatch is ECHL,
goto DOTN

if OI dispatch is ECHR, reset
OI dispatch to ECHL

if OI dispatch is neither ECHL or ECHR,
set OI dispatch to ECHR

copy ECHWD2 to ECHWD1

put new character in ECHWD2

restart output to the terminal
by calling OUNEW

return from ECHO

8.2.2.1.3 User command processing

8.2.2.1.3.1 Command interpretation

The subroutine ESC1 interprets local TIP commands. Each time through, ESC1 interprets one character which must be in the AC when it is called. The command status of each device is saved in a table called COMAND and converted to the temporary parameter COM when ESC1 is called. Only the first character of a word is significant; the command @ABORT LOGIN, for example, is equivalent to @A L. Extra spaces and the rest of each word are ignored by ESC1. By default, a command affects the device from which it was given. If, however, a command is preceded by a number (as in @2 A L, a command will affect the device so designated (in this case device 2). In this case the affected device is captured by the commanding device (which may be itself) and cannot be commanded by any other device until it is "given back" with a GIVE BACK or RESET command. In general, there are two kinds of commands, those with and those without parameters. Once the command is accepted and interpreted, ESC1 dispatches to specific code for each command. Parameters, if any, follow the command itself (e.g., @L 69) and are interpreted by this specific command code. Descriptions of these specific commands appear in the next section.

```
ESC1    initialize SKIPS, LFFLG; save character

        if mag tape command, goto MC

        look for a special character

        if character is not a space,
        goto ESC1a

        ignore character; reset flag

        return 1 from ESC1 to wait for next
        character
```

```
ESC1a     if character is a LF, goto RETURN

          if character is TELNET control, return 1
          from ESC1 to wait for next character

          if character is rubout, goto EXIT

          if character is lower case, convert
          it to upper case
```

```
Q2      look at status now

        if a space is needed, return 1 from ESC1
        to wait for next character

        if a command has already been matched,
        goto PAR

        if we are interpreting a command,
        goto SYNTAX

        if still looking for number, goto Q3

        if character is an esc, goto EXIT

        undivert output

        goto Q3


EXIT    clean up

        return 2 from ESC1 to indicate done
```

```
RETURN      if have nothing yet, goto EXIT

            if haven't got a dispatch yet,
            goto RETURNa

            set up dispatch address

            goto DISP


RETURNa     reset space flag, set LF flag

            goto SSYN1
```

PAR — set up parameter dispatch

goto DISPC

Q3 — if character is a number, goto Q3a

SSYN1 — correct status

goto SYNTAX

Q3a — accumulate a diverting device

set status to looking for number

return 1 from ESC1 to wait for next character

```
SYNTAX      interpret command syntax

Q6          pick up the next character
            to match

            if this one is not to
            be skipped, goto Q6a

            adjust SKIPs flag

            goto Q6



Qba         if not at end of
            tree, goto Q8

            if a parameter is needed, goto LFCHRa

LFCHR       if character is a LF, goto ERR1

LFCHRa      set up dispatch

DISPC       go do it

            goto EXIT



ERR1        send out "BAD"

            goto EXIT
```

```
Q8      if character is
        LF, goto Q9

        if character doesn't match, goto Q9

        save new status

        return 1 from ESC1 to
        get next character

Q9      if there is more to match,
        goto Q6

        goto ERR1
```

8.2.2.1.3.2 Command execution

Local commands interpreted by the TIP subroutine ESC1 cause the TIP itself to take the actions noted below. Most generally, commands affect either local terminal handling functions or network connection functions. Where parameters or other information are required it is so noted. The commands are listed in alphabetical order and cross reference other commands as appropriate.

ABORT LOGIN
Abort the outstanding initial connection protocol.

BINARY INPUT END
Leave 8-bit binary input mode; this command must be given from another device since commands are not recognized in binary input mode.

BINARY INPUT START
Enter 8-bit binary input mode.

BINARY OUTPUT END
Leave 8-bit binary output mode.

BINARY OUTPUT START
Enter 8-bit binary output mode.

CLEAR DEVICE WILD
Set device to be unwild; i.e., stop accepting RFC's from any Host.

CLEAR INSERT LINEFEED
Stop inserting linefeed after carriage-return.

CLOSE
Close all outstanding connections for this device.

DEVICE CODE 37
Establish parity computation for Model 37 Teletype for output to this device.

DEVICE CODE ASCII
Establish code conversion for an ASCII terminal.

DEVICE CODE EXTRA-PADDING
Establish code conversion for a terminal with slow CR; i.e., supply padding for correct carriage control.

DEVICE CODE OTHER-PADDING
Establish code conversion for a line printer; i.e., supply padding for correct carriage control.

DEVICE RATE #
is a 13-bit code specifying hardware rate and character size settings for the device commanded.*

DIVERT OUTPUT
Capture device # and divert this terminal's output to it.

ECHO ALL
Commands the TIP to perform local TIP-generated echo--TIP echoes everything.

ECHO HALFDUPLEX
Commands the TIP to expect terminal-generated echo--TIP echoes nothing.

ECHO LOCAL
Send the Telnet "ECHO LOCAL" character and perform an internal E A.

ECHO NONE
Commands the TIP to expect remote Host-generated echo for data--TIP echoes commands only and not data.

ECHO REMOTE
Send the Telnet "ECHO REMOTE" character and perform an internal E N.

FLUSH
Delete all characters in this device's input buffer.

GIVE BACK
Release control of captured device #.

HOST #
Simultaneous "@S T H" and "@R F H"; manual initialization of send and receive Host parameter to #.

INITIAL CONNECTION PROTOCOL
Start the initial connection protocol.

INSERT LINEFEED
Commands the TIP to insert linefeed after carriage-returns.

- - - - - - - - - - - - - - -

* # denotes a decimal number unless otherwise stated.

INTERCEPT #
Use # as TIP command character instead of 64 [i.e., @].

INTERCEPT ESC
Leave 7-bit binary mode; this command must be given from another device since a device in 7-bit binary mode ignores commands.

INTERCEPT NONE
Enter 7-bit binary mode; the TIP will ignore commands given by this device.

LOGIN #
Start the initial connection procedure with Host # to get Telnet connections--equivalent to "@S T H" followed by "@R F S 1" followed by "@I C P".

M # #
Mag tape command # with argument #.

NETWORK-VIRTUAL-TIP-EXECUTIVE
Connects the user to the Network-Virtual-TIP-Executive.

PROTOCOL BOTH
Simultaneous "@P T T" and "@P T R".

PROTOCOL TO CLOSE RECEIVE
Manually initiate protocol to close the receive connection.

PROTOCOL TO CLOSE TRANSMIT
Manually initiate protocol to close the transmit connection.

PROTOCOL TO RECEIVE
Manually initiate protocol to open a receive connection.

PROTOCOL TO TRANSMIT
Manually initiate protocol to open a transmit connection.

RECEIVE FROM HOST #
Establish Host # parameter for manual initialization of both send and receive Host.

RECEIVE FROM SOCKET #
Establish socket # parameter for manual initialization of the receive connection--socket # is given in octal.

RECEIVE FROM WILD
Equivalent to "@R F S <any>"; Host remains as specified.

RESET
Reinitialize a particular TIP port.

RESET NCP
Resets NCP of the Host parameter associated with this device.

SEND BREAK
Send the Telnet "BREAK" character.

SEND COMMAND
Send the command escape character.

SEND SYNC
Send the Telnet "SYNC" character and an "INTERRUPT SENDER" message.

SEND TO HOST #
Establish Host # parameter for manual initialization of the transmit connection.

SEND TO SOCKET #
Establish socket # parameter for manual initialization of the transmit connection--socket # is given in octal.

SEND TO WILD
Equivalent to "@S T S <any>"; Host remains as specified.

SET DEVICE WILD
Equivalent to the commands "@R F H <any>", "@S T H <any>", "@S T S <any>", and "@R F S <any>", and, in addition, leave these in effect after a connection is closed.

TRANSMIT EVERY #
Send off the input buffer at least every #th character where 0 <# <256.

TRANSMIT NOW
Send off the input buffer now.

TRANSMIT ON LINEFEED
Send the input buffer every time a linefeed is encountered.

TRANSMIT ON MESSAGE-END
Send the input buffer every time an end-of-message is encountered.

8.2.2.1.4 Rate initiation for "hunting" devices

The HUNT routine determines the rate of a device. Certain TIP ports may always have the same devices attached to them, such as line printers, and may be pre-initialized to the particular requirements of those devices. Other TIP ports, however, will be used by very simple devices or by different devices, as in the case of modems: these ports are called "hunting." The first character typed on such a device should be the hunt character for that device (see TIP User's Guide). Hunting devices are initially (and after an @R command) set to a nominal rate of 134.5 baud; the expected character will thus vary depending on the actual rate of transmission. The HUNT routine compares the first character with the various possibilities to determine the correct rate.

```
HUNT    if this is not really
        a hunting device, goto ESC2

        compare character with
        espected responses

        if no match, goto HUNTa

        set up the right
        rate & code

        adjust the table dispatch
        accordingly

        send a HELLO message
        to the device

        goto DOTN


HUNTa   wait for another character,
        try again

        goto DOTN
```

8.2.2.1.5 Input Conversion

8.2.2.1.5.1 EBCDIC to ASCII Conversion Logic

At present the TIP does only one input conversion, from EBCDIC to ASCII. This section discusses the logic for converting from EBCDIC to ASCII for four types of PTTC and four types of Correspondence model 2741 terminals.

The EBCDIC to ASCII code conversion is handled by the routine called IBMIN using the main conversion tables labeled TAB1 and TAB1P and the auxiliary tables labeled TAB2, TAB3, and DISPIN, along with some state information in the tables NN and M.

The TAB1 conversion table is for the correspondence EBCDIC to ASCII conversion and is 128 elements long. The first half of TAB1 is used for direct table lookup of the ASCII equivalents of the lower case correspondence EBCDIC characters, and the second half of TAB1 is used for the upper case correspondence EBCDIC characters. The individual elements in TAB1 contain either the direct ASCII equivalent of the EBCDIC character to be converted, a pointer to the auxiliary conversion table TAB3, or an offset with which to dispatch through the address table DISPIN. If the preceding character was a quote ("), the character looked up in TAB1 is used as the index for a further lookup in the retranslation table TAB2, or the character looked up in TAB1 is mapped into some other character in TAB1.

The TAB1P conversion table is used identically to TAB1 except for PTTC EBCDIC characters.

The detailed logic of IBMIN follows:

```
IBMIN     get input character
          and save it
          (in IBCHAR)

          if line is in output mode, go to IBMQ7

          get upper/lower case bit
          for terminal and append
          it to left of character;
          save result (in MODELT)

          get model of terminal
          and use it to select table
          to access (TAB1 for Correspondence,
          TAB1P for PTTC)

IBMQ1     pull character out of correct
          position in correct table

IBMQ2     save character (in IBDATA)

          if character special
          (>200), goto IBMQ6

          if last character received
          was a quote, goto IBMQ3

          increment head position
          counter

          get saved character
          (from IBDATA)

IBMQ9     pass character on

          return from IBMIN
```

```
IBMQ3     mark that last
          character received was
          not quote

          if character = 140, goto IBMQ10

          if character >140, goto IBMQ5

          if character >37, goto IBMQ4

IBMQ10    character input has no
          translation, skip it

          goto DOTN


IBMQ4     build pointer to access
          character in retranslation
          table (TAB2)

          goto IBMQ1


IBMQ5     if character >172, goto IBMQ10

          character was a lower case
          letter preceded by quote

          map character into control
          lower case letter

          goto IBMQ2
```

```
IBMQ6       if dispatch through DISPIN
            is called for (character >360),
            goto IBMQ8


            if character is a shift up
            character (character = 360),
            goto INUC


            character requires lookup in
            an auxiliary table


            build pointer to auxiliary table
            (TAB3) based on terminal model and type


            goto IBMQ1
```

```
IBMQ8       dispatch on character through
            DISPIN (to INBS, INLC, INNL, INTAB,
            INCC, INDQ, INLF, or INERR)

IBMQ7       If character is a circle-D, mark
            line in input mode

            goto IBMQ1Ø

INUC        prepare a 1

            goto INUCa

INLC        prepare a Ø

INUCa       jam Ø or 1 in NCASE

            goto IBMQ10
```

```
INLF    mark that last character
        was not quote

        increase null counter
        by a bunch (call UPNULL)

        pick up a linefeed

        goto IBMQ9
```

```
INDQ        complement mark indicating whether
            last character received was quote

            increment head position
            counter

            if this is not the second of a
            pair of double quotes, goto IBMQ10

            pick up a quote

            goto IBMQ9
```

```
INBS    if last character was not quote,
        goto INBS2

        clear mark that last
        character was quote

        goto IBMQ1Ø


INBS2   subtract one from head
        position counter if head
        not at left margin

        pick up a backspace

        goto IBMQ9
```

```
INNL    zero head position
        counter

        increase null counter
        (call UPNULL)

        mark that last character
        was not quote

        mark that carriage
        return received

        pick up carriage-return

        goto IBMQ9


INCC    if attempting to enter output mode,
        goto INC2

        mark that line is attempting to enter
        input mode

        restart output [to get circle-C sent]

        goto INC4


INC2    unblock output [clear NHOLD]

INC4    pick up a line feed

        goto IBMQ9
```

```
INTAB     mark that last character received
          was not quote

          increment null counter
          a little

          increment head position
          to next multiple of 8

          pick up tab

          goto IBMQ9
```

8.2.2.2 MLC Output (OUTINs)

CLKOI	swap buffers and restart OI
	put a "device Ø" at end of buffer
	save address of end of buffer in PSOI
	call OI
	return from CLKOI
OI	
OIL2	pickup next device number and dispatch to one of LCHAR, NCHC, NCHAR, ECHL, ECHR, or return from OI (only device Ø does the latter).

```
LCHAR    call LINBSY

         if in middle of typing
         something, goto LAST1

         if any ERROR2 errors are pending,
         goto SETOI2

         if any ERROR errors are pending
         (besides "CLOSED", "T", "R", or
         "memory bits"), goto SETOI

         if net data is waiting, goto NEWBUF

         if there are any other errors pending,
         goto SETOI

         if terminal is not EBCDIC, goto LCHAR3

         if terminal is not in output mode, goto
         LCHAR3

         mark as "turning around"

LAST8    set delay timer and
         send a circle-C

         goto OIL1


LCHAR3   mark terminal inactive

         if device is not high speed (>2400 baud),
         goto OIL1
```

increment number of allowed extra OIs

if this was not an "added" OI, goto OIL1

clear memory of extra OI [ONMOR]

goto OIL1

NEWBUF mark to send allocate

set up pointer to proper half of buffer and set up byte count

if not doing commands from net, goto NEWBUFa

set to dispatch to CCHAR

goto NEWBUFb

NEWBUFa call DIRCHK

if conversion or CR timing is required, set dispatch to NCHC

if conversion or CR timing is not required, dispatch to NCHAR

NEWBUFb dispatch to NCHC, NCHAR, or CCHAR

NCHC get next character

if character is a TELNET
character, goto SPBYTE

if code conversion is needed, call IBMOUT
or GENPAR, as appropriate

if special CR control is needed, call ODEC,
MEMRX, or SLOWCR, as appropriate

goto NCH2

```
NCHAR       get next character

            if character is a TELNET
            character, goto SPBYTE

NCH2        format character for MLC

            if not searching for a DM,
            goto NCH2a

            if not typing net traffic,
            goto NCH2a

            throw character away

            goto SFUDGE
```

```
NCH2a       send character to MLC

NCH3        if more output is to follow,
            goto OIL1

            set dispatch to LCHAR

OIL1        step to next OI
            device number

            goto OIL2
```

8.2.2.2.1 2471 Line Control

A) Keyboard is unlocked, TIP is in input mode accepting data (NDIR=NTURN=Ø). One of two things eventually happens:

1) A Circle-C comes in (because a CR or ATTN was typed). We immediately revert to input mode by marking the line in "output turning" (NDIR=1, NTURN=1). OI will notice this state and will send a Circle-C to the 2741. The 2741 will then eventually unlock its keyboard and it will then send a Circle-D which will cause the line to be back in input mode.

2) Some output comes along.
NBREAK is set and the line is held open. After an appropriate wait, we send a char of "all mark" and set the line to "input turning" (NDIR=Ø, NTURN=1). OI will then send a Circle-D and mark the line to be in output mode. Output may then proceed.

B) Keyboard is locked, output is in progress (NTURN=Ø, NDIR=1). One of two things may happen:

1) There is no more output to send. The line is marked as in "output turning". The same code as for A.1, above, will send a Circle-C, and the eventual Circle-D will put the line into input mode.

2) A Break comes in (user hit ATTN). NHOLD is set and then proceed as in B.1. NHOLD will prevent any output from happening (i.e., you can only go down the A.2 path if hold is clear). NHOLD is cleared by the input of a Circle-C.

```
DIRCHK    if device is not EBCDIC,
          return from DIRCHK

          if device is in output mode,
          return from DIRCHK

          send reverse break and mark
          device as "turning around"

          goto OIL1
```

LINBSY if device is not EBCDIC,
return from LINBSY

if is not turning around,
holding off output, or
timing a reverse break, return
from LINBSY

if device not in "input turning
around to output" mode,
goto LINBSYa

mark in output mode and
send a circle-D

goto OIL1

LINBSYA if timing a reverse break
or holding off output,
goto LCHAR3

now turning from output to
input

if delay time ≠ Ø, goto LCHAR3

goto LAST8

MIFAST If device is synchronous or its output note is ≤ 2400 baud, return

set up INDMAX to the address of a word containing the number of parallel OIs required to achieve full speed

give skip return

FASTER if device is not high-speed, return

if this is already an "added" OI goto RESMER

if an OI has already been added during this pass through OI, return

if device is already operating at full speed, return

count one less OI needed for full speed

add an extra OI for this terminal on the end of the OI table

return

RESMOR clear memory that an OI has been added

return

8.2.2.2.2 TELNET Protocol Handler

SPBYTE if in binary output mode, goto NCH2

if character is an IAC, goto SPIAC

if character is "command from net", goto NETCOM

if character is not an "echo remote", goto SPBYTEa

call FIXECH(FDX)

goto SFUDGE

SPBYTEa if character is not an "echo local", goto SPBYTEc

call FIXECH(HDX)

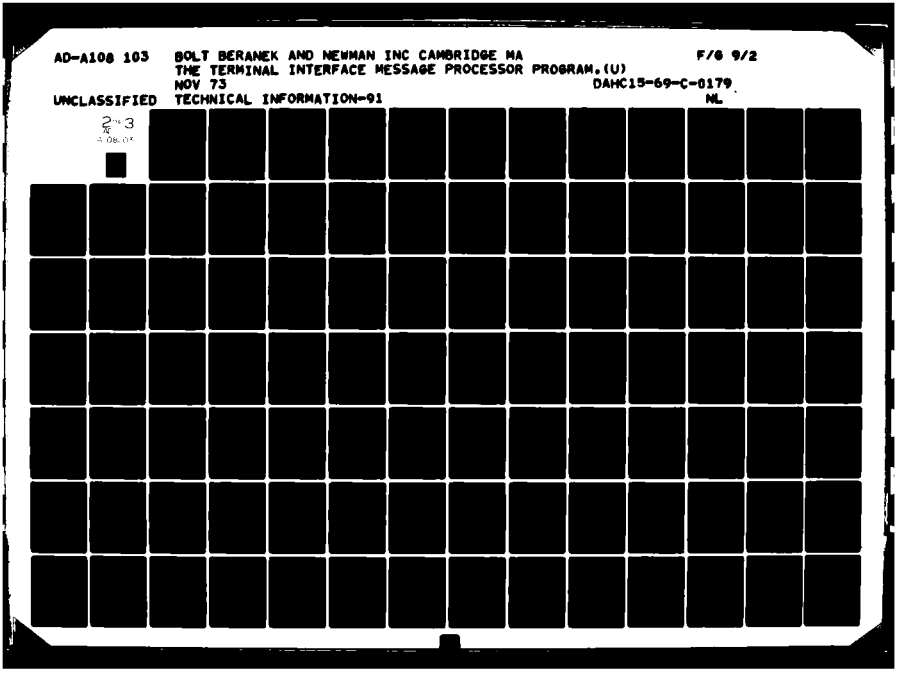
goto SFUDGE

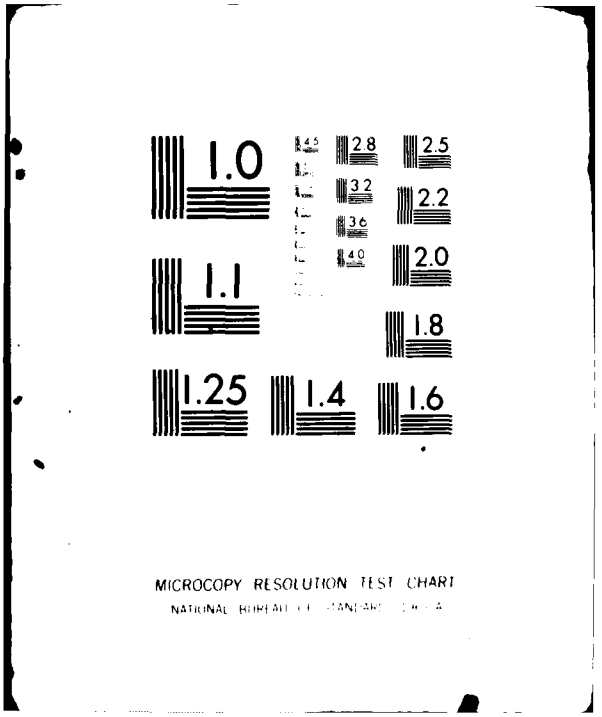
SPBYTEc if character is not a DM, goto SFUDGE

SPBYDM bump INS/DM count

SFUDGE set OI to reprocess this device (i.e., step to next character)

goto NCH3


AD-A108 103
BOLT BERANEK AND NEWMAN INC CAMBRIDGE MA
F/G 9/2
THE TERMINAL INTERFACE MESSAGE PROCESSOR PROGRAM,(U)
NOV 73
DAHC15-69-C-0179
UNCLASSIFIED
TECHNICAL INFORMATION-91
NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

```
SPIAC       get next byte

            if next byte is WILL, goto WILL

            if next byte is DO, goto DO

            if next byte is DM, goto SPBYDM

            otherwise, goto SFUDGE


WILL        if no room in buffer, throw
            away next byte and goto SFUDGE

            put IAC DONT in input buffer

            goto SPIACa


DO          if no room in buffer, throw
            away next byte and goto SFUDGE

            put IAC WONT in input buffer

SPIACa      copy next byte into
            input buffer

            call SENDIT

            goto SFUDGE
```

```
NETCOM      set "commands coming from
            net" bit

            reset dispatch to
            CCHAR

            dispatch to CCHAR

CCHAR       get next byte

            if no more, goto CCHARa

            if byte is CR, goto CCHAR

            pass character to command
            processor

            if command not completed,
            goto CCHAR

            clear "commands from net"
            bit

CCHARa      reset dispatch to LCHAR
            and dispatch to LCHAR
```

8.2.2.2.2.1 FIXECH

The following routine, FEXECH, is also called by the code for the user commands which set the echoing mode; however, the inclusion of the routine with the preceeding TELNET echo command code seems as natural as including it with the command code.

```
FIXECH(mode)    if input is not currently
                going to network, return from FIXECH

                set new input dispatch
                from mode

                return from FIXECH
```

8.2.2.2.3 Echoes

```
ECHL    call LINBSY

        call DIRCHK

        reset DISPATCH to ECHR

        send ECHWD1

        goto OIL1


ECHR    call LINBSY

        call DIRCHK

        reset dispatch to LCHAR

        send ECHWD2

        goto OIL1
```

8.2.2.2.4 Messages to devices

The routines SETOI and SETOI2 are used to detect if there are any bits set in ERROR (SETOI) or in ERROR2 (SETOI2). If a bit is set, a pointer and byte count for the associated error message are retrieved and passed to the terminal output routine. SETOI and SETOI2 are called by the MLC clock interrupt routine. The only thing less than straightforward about these routines is the handling of the ERDED2, EROPN2, and ERCLS2 bits by SETOI. These bits are where memory is kept of the fact that a "DEAD", "OPEN", or "CLOSE" message was recently printed, and are cleared when there are no "real" bits left in ERROR. Also, if both the "T" and "R" bits are set simultaneously, both are cleared. Of course, the ERDED2, EROPN2, and ERCLS2 bits are prevented from themselves causing a message to be printed since they actually represent no message.

ERDED2 is set by the routine DEAD when either a "HOST DEAD" or "NET TROUBLE" message is set to be printed. Its purpose is to cancel the "REFUSED" message to prevent "DEAD REFUSED" when logging to a dead Host, if the "REFUSED" comes soon enough after the "DEAD".

EROPN2 and ERCLS2 are set by ERRTEL to remember when an "OPEN" or "CLOSED" was recently printed, in order to prevent the "OPEN OPEN" or "CLOSED CLOSED" message when "CLOSED" or "OPEN" is printed but "R" and "T" are canceled because both are there, but they didn't arrive together so another "CLOSED" or "OPEN" is present.

SETOI build pointer to base of correct table depending on whether error is mag tape error or not

if both "T" and "R" bits are not set in ERROR, goto SETOIa

clear both "T" and "R" bits from ERROR

goto LCHAR

SETOIa if no bits are set in ERROR besides ERDED2, EROPN2, and ERCLS2 "memory bits", goto SETOIB

get the first non-memory bit which is set

if no other bits are set, goto SETOIc

SET6 get a pointer to the associated error message and its byte count

goto LAST2

SETOIb clear ERDED2, EROPN2, and ERCLS2 "memory bits"

goto LCHAR

```
SETOIc      clear first bit in ERROR
            which was found set

            goto SET6


SETOI2      build a pointer to correct
            table for ERROR2 errors

            find the first bit
            which is set in ERROR2

            clear bit

            goto SET6
```

The three routines ERRTEL, HOLLER, and HOLL2 are used to set flags which the terminal output routines detect and then print the indicated message on the TIP terminal. Two words of message flags are associated with each device; these are in the tables ERROR and ERROR2. HOLLER sets flags in ERROR; HOLL2 sets flags in ERROR2. ERRTEL is called rather than HOLLER for those messages which may have an associated "T" or "R", e.g., "OPEN T". The handling of the "CLOSED" and "OPEN" messages by ERRTEL is a little complicated. If either of these bits is set in the call to ERRTEL, the associated "memory bit," ERCLS2 or EROPN2, is set to remember that "CLOSED" or "OPEN" was just printed. Making use of this memory, if ERCLS2 or EROPN2 is set when ERRTEL is called with a "CLOSED" or "OPEN", that bit is cleared to prevent two "CLOSEDs" or "OPENs" from being printed in a row. This is done by clearing the "OPEN" or "CLOSED" bit in the argument to ERRTEL if the associated memory bit is set before the argument is passed to HOLLER.

HOLLER OR bits in argument
to HOLLER into ERROR

set CR-LF bit in ERROR

return from HOLLER

HOLL2 OR bits in argument to
HOLL2 into ERROR2

call HOLLER with Ø argument
to set CR-LF bit in ERROR

return from HOLL2

```
ERRTEL(X)  if neither CLOSED-bit or
           OPEN-bit set in X, goto ERRTELa

           for whichever of the CLOSED or
           OPEN bits is set in X, set the associated
           memory bit, ERCLS2 or EROPN2, in X

           for whichever of the CLOSED
           or OPEN bits was set in X, if the associated
           memory bit, ERCLS2 or EROPN2, is set,
           clear the CLOSED or OPEN bit in X

           goto ERRTELb


ERRTELa    if ERRTEL was called by a "T
           routine," OR T-bit into X

           if ERRTEL was called by an
           "R routine," OR R-bit into X

ERRTELb    call HOLLER with argument X to
           set bits in ERROR

           return from ERRTEL
```

8.2.2.2.5 Output conversion

8.2.2.2.5.1 ASCII to EBCDIC Conversion Logic

This section discusses the logic of converting from ASCII to EBCDIC (TIP to terminal) for four types of PTTC and four types of Correspondence model 2741 terminals.

The ASCII to EBCDIC code conversion is handled by the routine called IBMOUT using the main conversion tables labeled CVTB and CVTBP and the auxilary conversion table SCVTB, along with some state information in the tables NN and M.

The CVTB conversion table is for the ASCII to Correspondence EBCDIC conversion and is 128 elements long, suitable for direct indexed access by the ASCII character to be converted. The individual element in CVTB either gives the direct conversion to a Correspondence EBCDIC character, indicates that a quote (") character should be printed before the Correspondence EBCDIC character also given, or indicates an entry in the auxiliary table SCVTB which contains the proper Correspondence EBCDIC character and perhaps further indications. Each entry in the auxilary table SCVTB has four subelements, one for each of the four types of Correspondence terminals. The CVTB entries also indicate the proper case for the Correspondence EBCDIC character to be printed.

The CVTBP table is identical in form and use to the CVTB table except that it is used for ASCII to PTTC EBCDIC conversions.

The detailed logic of IBMOUT follows:

```
IBMOUT    save ASCII character to
          be converted (in CHARIB)

          if the previous character printed
          was a carriage-return (MGOTR≠ 0), goto PØ3

PØ4       get base of proper conversion table,
          CVTB for Correspondence, CVTBP
          for PTTC

          build pointer to converted character

PØ2       save pointer (in IBMT2)

          get converted character
          and save it (in HOLDIT)

          if auxiliary lookup is to be done
          and if character not to be preceded
          by quote, goto PØ1

          if character is to be preceded
          by quote, goto P11

P13       if case matters for character
          to be printed, goto PØ9

          mark that quote was not last
          character sent (NQUOTE←Ø)

          increment head
          position counter

          get character to be printed
          (from HOLDIT); mask character
          down to 7 bits

          goto NCHA1 (to print character)
```

P09 if terminal's case is already correct, goto P08

if case should be upper case, goto P10

complement case (NCASE←$\overline{\text{NCASE}}$)

goto NCHA0 (to print lower case character)

P10 complement case (NCASE←$\overline{\text{NCASE}}$)

goto NCHA0 (to print upper case character)

P11 if the last character printed
was a quote (NQUOTE ≠ Ø), goto P13

if terminal's case is already
upper case, goto P1Ø

increment head
position counter

mark that last character sent
was quote (NQUOTE←1)

build a proper quote
character for model and type 2741

goto NCHAØ (to print quote)

PØ1 if character to be converted
is an ASCII carriage-return, linefeed,
or tab, goto PØ1A

if character to be converted is
an ASCII backspace, goto OIBS

build pointer to the auxiliary table
SCVTB based on the terminal
type and the character to be converted

goto PØ2

P03 if the character to be converted is a linefeed, goto P05

if the head position is not zero, goto P14

mark that last character sent not a carriage-return (MGOTR←0)

goto P04

P14 decrement terminal head position

goto NCHA0 (to print backspace)

P05 if enough nulls have been sent to terminal (NNULL = 0), goto P15

P17 decrement count of nulls to be sent to terminal (NNULL←NNULL - 1)

if enough nulls have not yet been sent to terminal, goto P06

mark that last character sent was not a carriage return

goto NCHA4 (to send a null to terminal)

P15 clear head position counter
after first noting its value

if head was so far out extra
nulls need to be sent, goto P15a

P15b goto NCHAØ (to print a new line)

P15a put large value in counter
of nulls to be sent

goto P15b

PØ6 goto NCHAØ (to send a
null terminal)

PØ1A if character to be printed
is a tab, goto OITAB

if character to be printed
is a linefeed, goto OILF

mark that last character printed
was a carriage return (MGOTR←1)

goto SFUDGE

OIBS if head position counter $\neq \emptyset$, goto OIBSa

OIBSb goto NCHA4 (to print a backspace)

OIBSa move head position counter back

goto OIBSb

OILF if there is not a null character to be sent to the terminal, goto P17

mark that a null character should be sent to terminal

goto NCHAØ (to print a linefeed)

OITAB if there is null character to be sent to the terminal, goto P17

increment the head position counter by a bunch

increment number of null characters to be sent to terminal by a bunch

goto NCHAØ (to print a tab)

8.2.2.2.5.2 Slow Carriage Return

SLOWCR save character to be printed (in SLOWCH)

if a carriage return just sent, goto SLOWCRa

if this character is a carriage-return, goto SLOWCRb

SLOWCRc add 1 to head position

goto NLNA4 (to print character)

SLOWCRb mark that carriage-return just sent

add a large constant to the head position

goto SLOWCRc

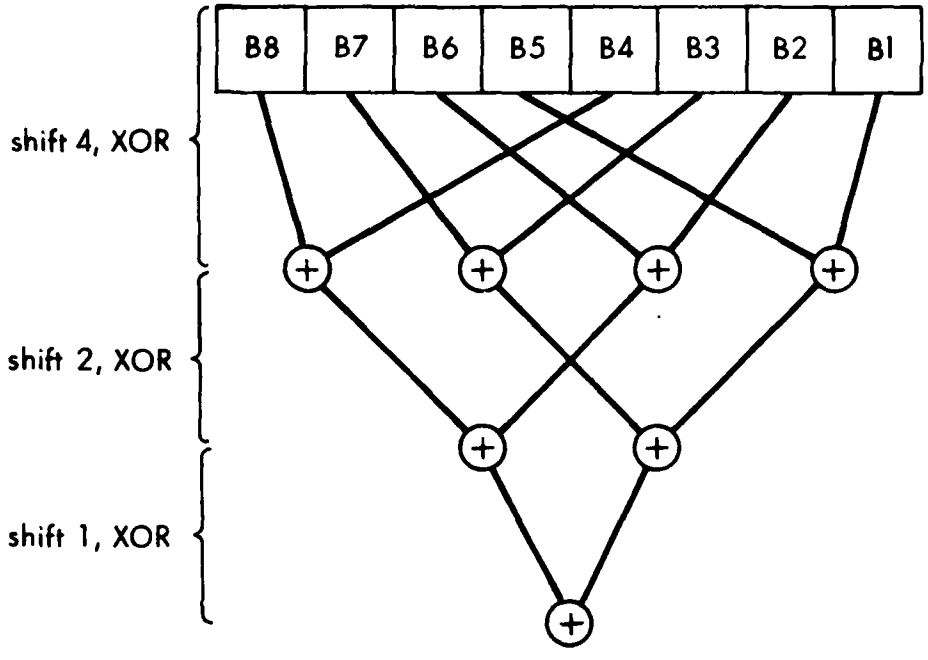
```
SLOWCRa   build a constant which is
          function of device rate

          subtract this constant from
          head position

          if head is at
          left margin, goto SLOWCRd

SLOWCRe   update head position

          if this character is a
          linefeed, goto SLOWCRc

          goto NCHAØ (to print rubout)


SLOWCRd   zero head position

          clear just sent
          carriage-return bit

          goto SLOWCRe
```

8.2.2.2.5.3 Even Parity

The even parity calculation routine, GENPAR, calculates a character's parity with three successively smaller shifts followed by exclusive-or's. The effect of this is to successively "fold" the character on itself until only the character's parity is left. The parity is then appended to the original character and the character with parity is passed to NCH4 for dispatch to the correct carriage-return padding routine. The logic of GENPAR is then, simply:

```
GENPAR    save character
          compute parity of character
          append parity to character
          goto NCH4
```

For help in reading the GENPAR code, the character folding takes place as follows.


B8
B7
B6
B5
B4
B3
B2
B1
shift 4, XOR
shift 2, XOR
shift 1, XOR

8.2.2.2.5.4. Line Printers

Line printers connected to the TIP frequently require special padding characters after carriage-returns, form feeds, etc. The TIP possesses code to support two different brands of line printers, the ODEC printer and the Memorex printer, although a given TIP can only be configured with the code for one or the other but not both.

8.2.2.2.5.4.1 MEMORX

The MEMORX routine is the output dispatch that handles carriage control requirements of the Memorex 1240 Communications Terminal (and any other compatible device). The padding is tailored to a rate of 600 baud. The "rules" follow:

1) a line must be at least 43 characters long; if it is not, padding is added before sending the carriage return;

2) contiguous line feeds must be separated by 3 characters;

3) a vertical tab is followed by 27 padding characters;

4) a form feed is followed by 215 padding characters.

```
MEMRX     if not outputting character, goto MEMNUL

          if character is LF, goto MEMRXa

          if character is CR, goto MEMRXb

          if character is FF, goto MEM4

          if character is VT, goto MEM4

MEMSVI    decrement line count;
          output character

          goto NCHA4


MEMNUL    if finished with padding, goto MEMA1

          decrement count; output rubout

          goto NCHAØ


MEMRXa    if last character was a LF, goto MEM4

          mark it LF

          goto MEMSV1
```

MEM4 set up padding count

goto MEMNUL

MEMRXb if line is not already long enough, goto MEM4

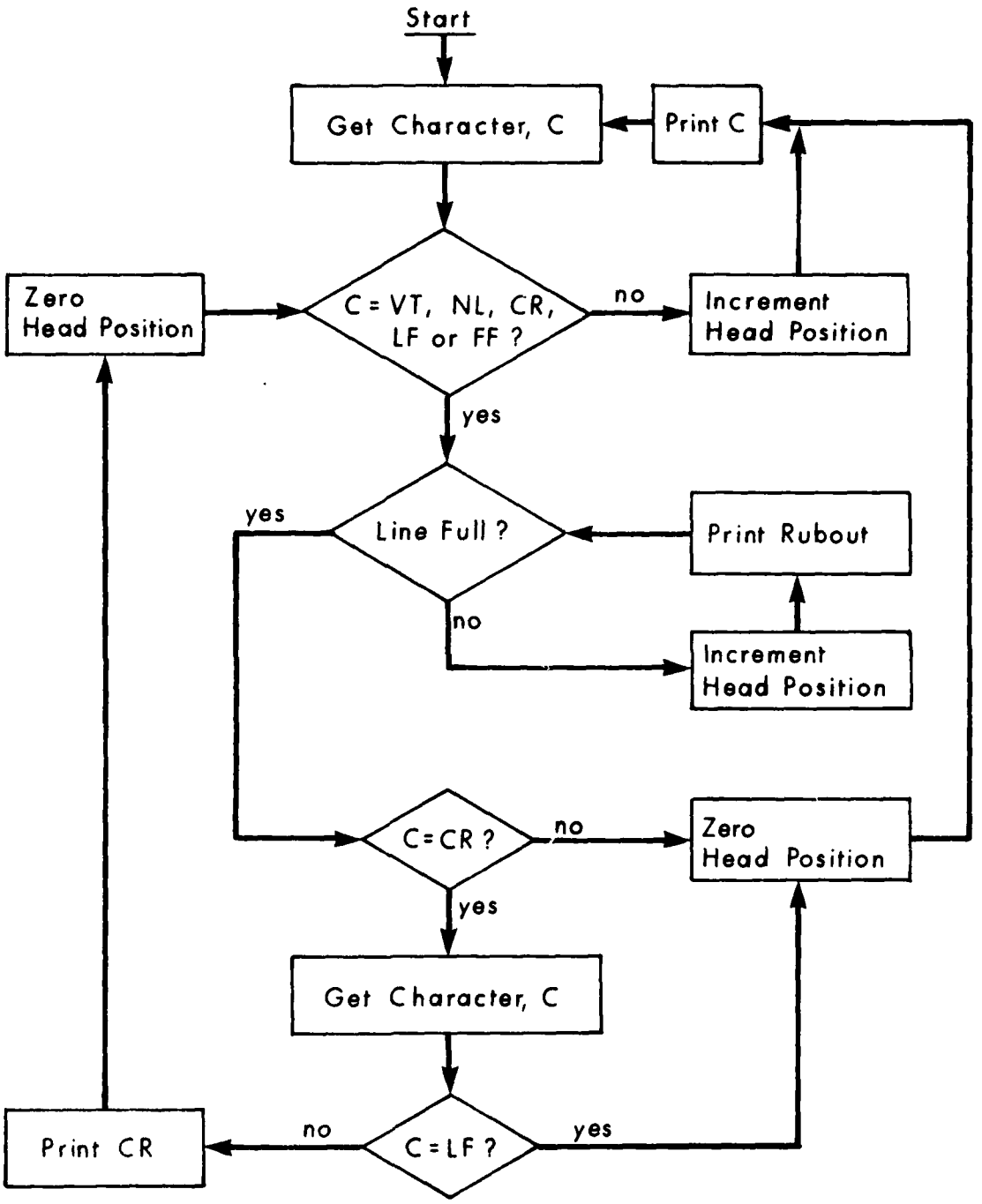
MEMRXc set up count for new line

goto MEMSV1

MEMA1 if character is not control character, goto MEMRXc

set up appropriate count

goto MEMSV1


Start
Get Character, C
Print C
Zero Head Position
C = VT, NL, CR, LF or FF ?
no
Increment Head Position
yes
yes
Line Full ?
Print Rubout
no
Increment Head Position
C = CR ?
no
Zero Head Position
yes
Get Character, C
Print CR
no
C = LF ?
yes

8.2.2.3 Background Timer (BTIME)

The background timer routine, BTIME, is called by the MLC clock interrupt routine to print nulls until the null counter is zero and then, if the device is a 2741 which is reverse breaking, to set the line turn-around flag.

```
BTIME       if one (2741) character time has not gone
            by, return from BTIME

            i←Ø

BTIME2      if output is in progress
            to device i, goto BTIMEa

            if there are
            sent to device i, goto BTIMEa

            decrement null counter

            if device i is not a 2741,
            goto BTIMEa

            if device i is not "reverse
            breaking", goto BTIMEa

            clear "reverse breaking" flag

            set "line turn around" flag

            call IBECHO to print null

BTIMEa      i←i+1

            if i ≠ 64, goto BTIME2

            return from BTIME
```

8.2.3 Background

The TIP background loop, BACK, is quite straightforward as is shown below; BACK is called by the IMP background loop.

```
BACK    if TIP should be initialized
        (TIPFLG ≥ Ø), goto BACKa

        call SLURP

        call MODEM C

        call BCKWAK

        call SENDW

        return from BACK

BACKa   mark that initialization is no
        longer needed (TIPFLG←-1)

        call GOGGLE

        return from BACK
```

The routines BCKWAK and BCKSLP have a co-routine relationship. BACK calls BCKWAK to restart BCKSLP (and its subordinate routines) where it last left off. BCKSLP subordinate routines call BCKSLP to wait (sleep) until the next pass through BACK. BCKSLP is initially set to restart at BCKCHK.

BCKWAK return from BCKSLP (to restart BCKSLP where it last left off)

BCKSLP return from BCKWAK

BCKCHK call LOGGER

call PROBCK

call PENDIN

if appropriate, call MTBACK

call BCKSLP

goto BCKCHK

8.2.3.1 Connection and Host Functions (PENDIN)

PENDIN is a subroutine called by the background loop. It checks all devices in round-robin fashion to see if they have pending any of the following: one, data to go to the net; two, an allocate to be sent; three, miscellaneous control messages such as interrupts, resets, echo reply, etc.; four, a request for news; five, replies to unsolicited messages. PENDIN serves as a wakeup for these jobs and takes appropriate action when it finds something to do. As a general comment, the TIP packs control messages going to the same Host into the same Host/Host protocol message. Using this feature, PENDIN continues to check all devices after it has set up a control message and appends any others going to the same Host.

```
PENDIN    if there is nothing to do,
          return from PENDIN

          init to the next device
          in round-robin order

PEND1     set up dispatch table
          based on flags; entry
          is NOP or JMP

PEND3     if entry is JMP, goto PENDA

PEND4     if entry is JMP, goto PENDB

PEND5     if entry is JMP, goto PENDC

PEND6     if entry is JMP, goto PENDE

PEND8     inc. to next device

          if not finished cycling through all
          devices, goto PEND3

          goto PENDUN
```

PEND7 send off any messages to the IMP with a call to FIRE

return from PENDIN

8.2.3.1.1 Device Related functions

8.2.3.1.1.1 Transfer data to IMP

```
PENDA   if device does not have data to go,
        goto PEND4

        call SENDIT and SENDW to try to get
        it going

        goto PEND4
```

8.2.3.1.1.2 Allocates

PENDB if there is no need to send an allocate for device, goto PEND5

if device is not being diverted to, goto PENDB1

use capturing device's buffers for ALLOC

PENDB1 if connection is closing or closed, goto PEND5

if connection not open yet, goto PENDB2

call SENDAB to try get the interface

if did not get interface, goto PENDB2

set up an allocate message

mark allocate as sent

goto PEND5

```
PENDB2      set bit to try
            device later

            goto PEND5
```

8.2.3.1.1.3 News

```
PENDE     if not trying to
          get news, goto PEND8

          if tried all news
          Hosts, goto PEND8

          call SENDAB to try
          to get interface

          if did not get interface
          goto PEND8

          send an RFC to
          a news Host

          set up next Host
          in list for next time

          goto PEND8
```

8.2.3.1.2 Host related functions

8.2.3.1.2.1 Send INS's

```
PENDC     if don't need to
          send interrupt, goto PENDD

          if connection not
          open, goto PENDD

          call SENDAB to try get
          the interface

          if did not get interface,
          goto PENDD

          set up an interrupt message

          goto PENDD
```

8.2.3.1.2.2 Send ERP's, RST's, and RRP's

PENDD call PENDD to send ERP, RRP, RST if needed and can for transmit side

call PENDH to send ERP, RRP, RST if needed and can for rcv. side

goto PEND6

PENDH call SNDRP to try to send ERP's

call SNDRP to try to send RRP's

call SENDRP to try to send RST's

return from PENDH

```
SNDRP(A)  if Host is not marked as
          needing A sent, return from SNDRP

          set up Host to send to for
          device Ø1

          call SENDAB to try to get
          interface to IMP

          if did not get interface, return from SNDRP

          clear mark for Host saying A
          should be sent

          if A=ERP, goto SNDRPa

          for all devices talking to this Host, close
          the connection if not already closed (calls
          SHUTDN) and print "CLOSED"

          build an A (RST or RRP) in IMP buffer
          with call to PUTLST

          return from SNDRP


SNDRPa    build an ERP in IMP buffer with
          call to PUTLST

          return from SNDRP
```

8.2.3.1.2.3 Send CLS's for unsolicited RFC's and CLS's

```
PENDUN    if a CLS for an unsolicited
          RFC or CLS has not been saved,
          goto PEND7

          call SENDAB to get interface

          if did not get interface, goto PEND7

          clear the saved CLS just sent

          goto PEND7
```

8.2.3.2 Connection Control (PROBCK)

PROBCK is a subroutine called by the background loop. It checks the status of all connections (active or not) and sends "open" or "close" messages to Hosts if required. PROBCK uses the two connection tables PSTATE and QSTATE, respectively send and receive, which indicate status as follows:

0 = try to open
1 = RFC sent
2 = RFC received, try to reply
3 = solid connection
4 = try to close
5 = CLS sent
6 = CLS received, try to reply
7 = no connection

```
PROBCK    if flag not set, return from PROBCK

          init counter; init index
          to lst connection

          goto PROB2


PROB3     increment counter; increment index

          if not done all connection, goto PROB2

          return from PROBCK


PROB2     if this connection does not
          need doing, goto PROB3

          set flag to do PROBCK again

          if transmit connection and data
          link blocked, goto PROB3

          if receive connection and output
          active, goto PROB3

PROB1     try to get Host for
          control msg with call
          of SENDAB

          if did not get interface, goto PROB3

          if connection trying to close, goto PRO3

          connection trying to open

          if rcv. side, goto PRO3a
```

```
                transmit side, TRYPRT←10

                AC←code for STR

                goto PRO5


PRO3            TRYPRT←0
                AC←code for CLS

                goto PRO5


PRO3a           TRYPRT←dev + 2

                AC←code for RTS

PRO5            set up msg to Host

                if not trying to close, goto PRO5a

                call SHUTDN to close the connection

                return from PROBCK


PRO5a           call FIRE to send the message

                return from PROBCK
```

8.2.3.3 Initial Connection Protocol (Logger)

The TIP's logger can do the Initial Connection Protocol for only a single device at a time. Device requests to log are queued via the MDLOG bit. When the logger accepts a device, it will ICP to HOSTS over SOCKR1 and SOCKR2(!). To keep track of what it is doing, the logger uses two three-state flags, LOGS and LOGOPN. LOGS saves the internal state of the logger, while LOGOPN reflects the state of the remote logger.

The three states of LOGS are:

1) =Ø LOGGER IS IDLE
2) =1 LOGGER HAS ATTEMPTED TO OPEN LOGGING CONNECTION
3) >1 LOGGER HAS ATTEMPTED TO CLOSE LOGGING CONNECTION

The three states of LOGOPN are:

1) =Ø LOGGING CONNECTION NOT MADE
2) <Ø LOGGING CONNECTION IS OPEN
3) >Ø LOGGING CONNECTION IS CLOSED

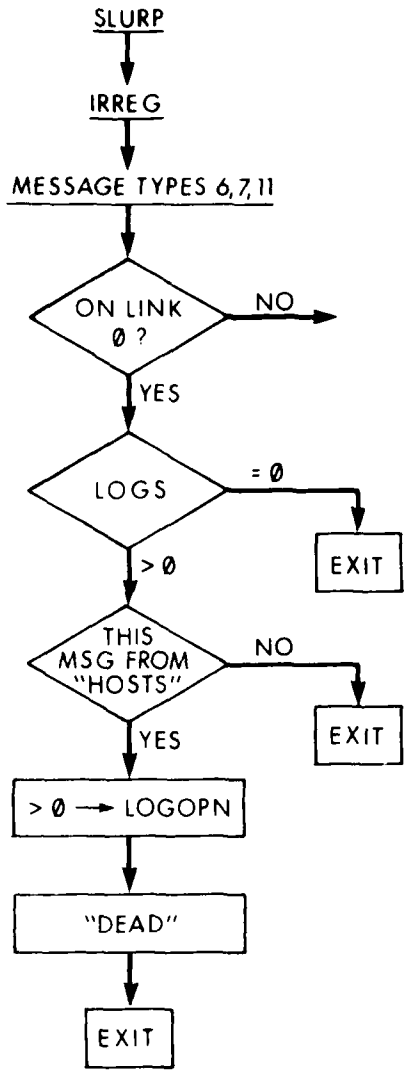
LOGOPN is set by TESTOK (from GETCLS, GETSTR, GETRTS; from PROT; from SLURP).

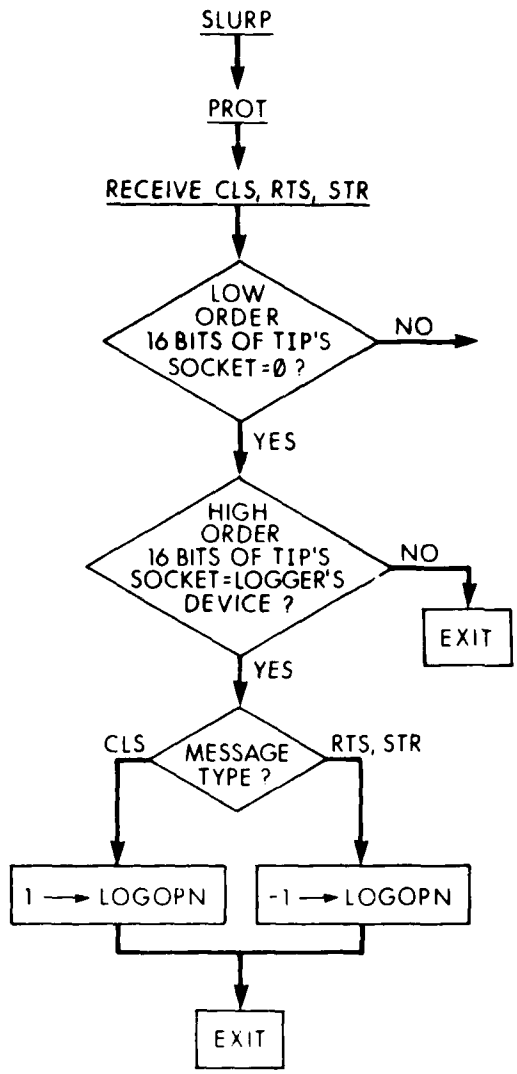
The logger attempts to get the logging connection open and closed as quickly as it can, ignoring any data that comes in over the connection. Once the logging connection is dispensed with, the device is set wild and then the logger drops off the device.

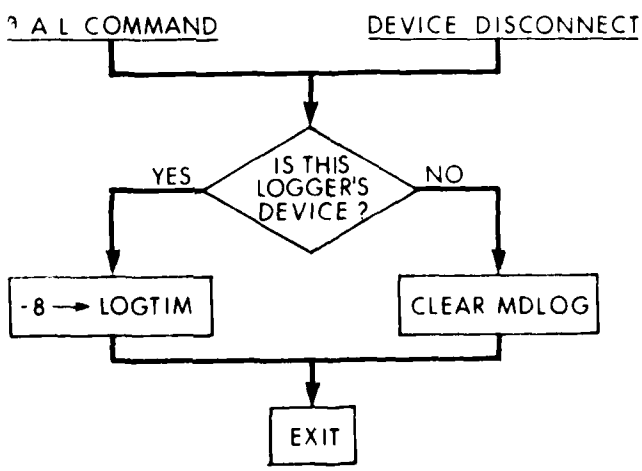
The logger allows 5ØØ "logger ticks" (≈ 18ØØ MLC clock ticks) to complete its part of the ICP. Note that this time only runs from the beginning of the login sequence up to the closing of the logging connection. The device must fend for itself on getting

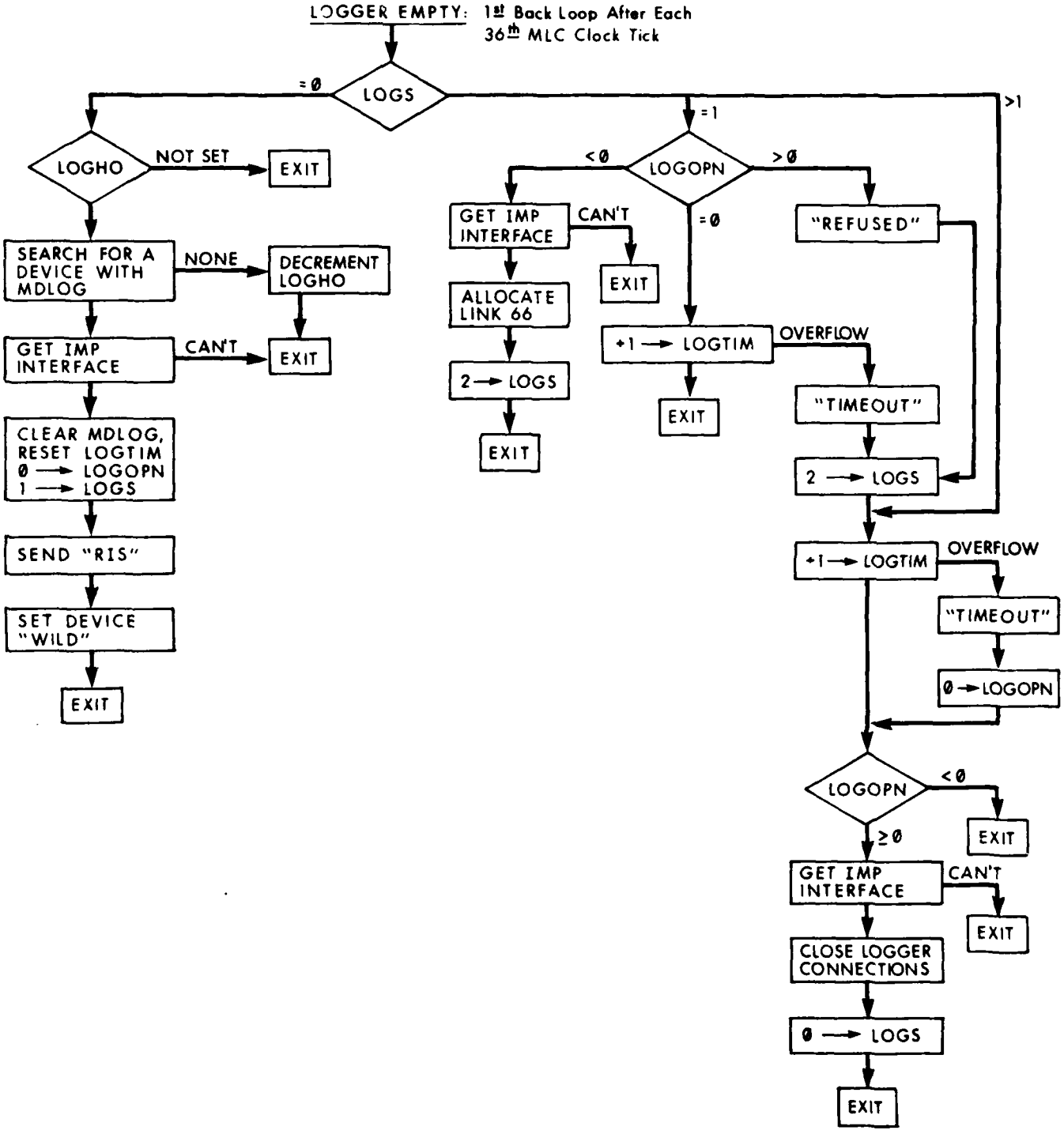
the data connections open, and the user must fend for himself waiting to get to the logger. All "abort login' does is to set the timer close to overflowing, thus doing nothing if the user is not yet on, or is already off, the logger. Login is also aborted by the disconnect code.

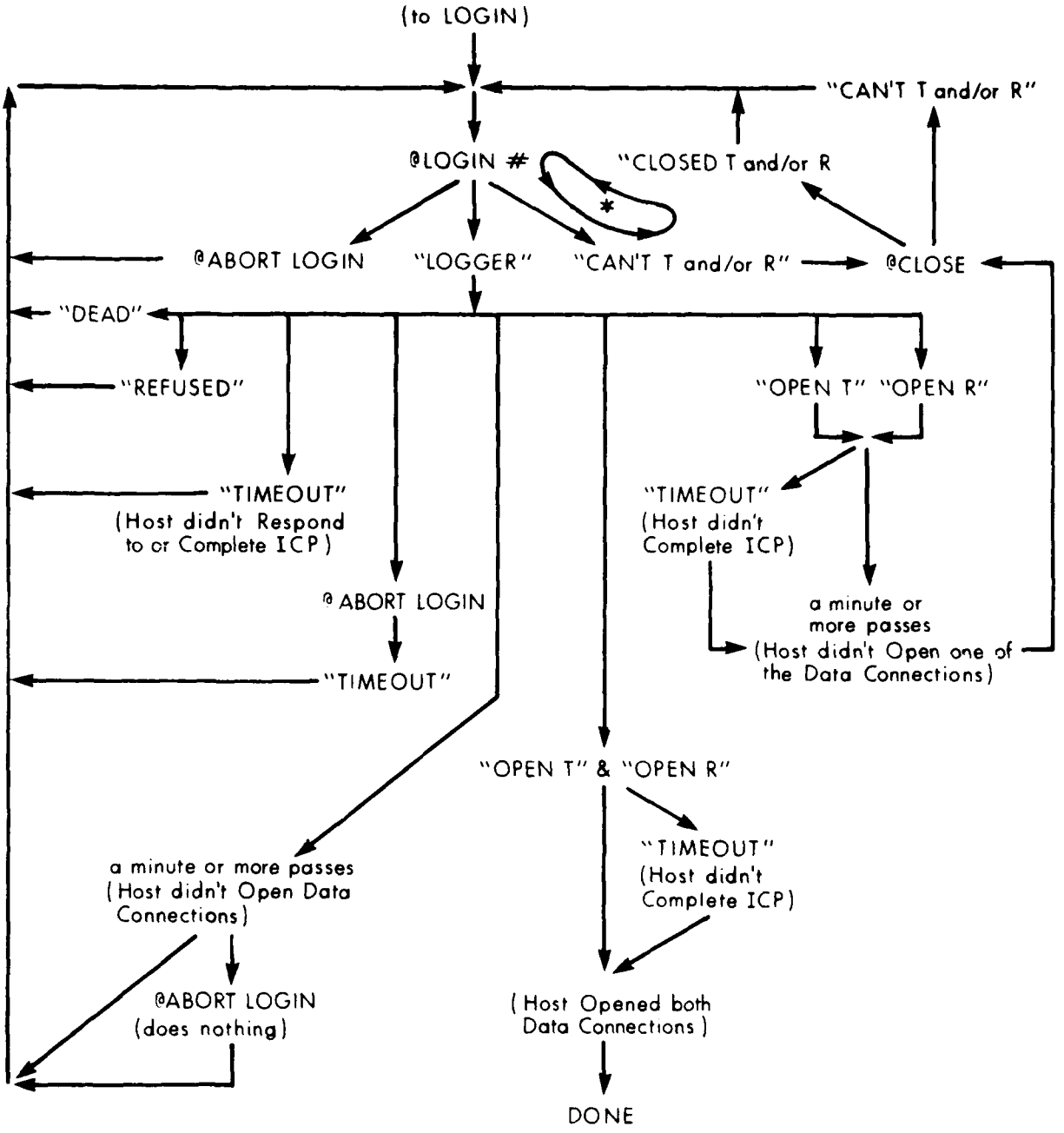
There follow some diagrams of portions of the logger logic, followed by a chart giving the logic as seen by the user.


SLURP
IRREG
MESSAGE TYPES 6,7,11
ON LINK Ø ?
NO
YES
LOGS
= Ø
EXIT
> Ø
THIS MSG FROM "HOSTS"
NO
EXIT
YES
> Ø → LOGOPN
"DEAD"
EXIT


SLURP
PROT
RECEIVE CLS, RTS, STR
LOW ORDER 16 BITS OF TIP'S SOCKET=Ø ?
NO
YES
HIGH ORDER 16 BITS OF TIP'S SOCKET=LOGGER'S DEVICE ?
NO
EXIT
YES
CLS
MESSAGE TYPE ?
RTS, STR
1 → LOGOPN
-1 → LOGOPN
EXIT


? A L COMMAND
DEVICE DISCONNECT
YES
IS THIS LOGGER'S DEVICE ?
NO
-8 → LOGTIM
CLEAR MDLOG
EXIT


LOGGER EMPTY: 1st Back Loop After Each 36th MLC Clock Tick
LOGS
=0
=1
>1
LOGHO
NOT SET
EXIT
SEARCH FOR A DEVICE WITH MDLOG
NONE
DECREMENT LOGHO
GET IMP INTERFACE
CAN'T
EXIT
CLEAR MDLOG, RESET LOGTIM
0 → LOGOPN
1 → LOGS
SEND "RIS"
SET DEVICE "WILD"
EXIT
LOGOPN
<0
>0
=0
GET IMP INTERFACE
CAN'T
EXIT
ALLOCATE LINK 66
2 → LOGS
EXIT
+1 → LOGTIM
OVERFLOW
EXIT
"REFUSED"
"TIMEOUT"
2 → LOGS
+1 → LOGTIM
OVERFLOW
"TIMEOUT"
0 → LOGOPN
LOGOPN
<0
EXIT
≥0
GET IMP INTERFACE
CAN'T
EXIT
CLOSE LOGGER CONNECTIONS
0 → LOGS
EXIT



* It may take many seconds to get out of this loop

8.2.3.4 Send data to IMP

TILEAD store second leader word in IMP buffer

set up destination in first word
in IMP buffer

mark as not last
packet of message

fake interrupt of Host to IMP

return from TILEAD

8.2.3.4.1 Link Ø

SENDAB if SENDAB is already using interface (GOING≥ Ø), goto SENDABa

if anything else (e.g., data) is already using interface (TIGOF ≠ Ø), return 1 from SENDAB

if link blocked to destination Host, return 1 from SENDAB

block link

GOING←-1

SEND6 for all devices clear send-allocate-to-Host-bit as unblocked control link indicates no un-RFNMed allocates to this Host

call TILEAD to send leader without last packet bit set

SENDABb if IMP is not ready for more (TIGOF <-1), call BCKSLP to wait a while and then goto SENDABb

call PUT2 to fill in first three words of extended Header

return 2 from SENDAB

```
SENDABa   if to correct destination, return
          2 from SENDAB

          return 1 from SENDAB


FIRE      if GOING <Ø, return from FIRE

          GOING ← -1

          put padding in IMP buffer

          fake interrupt to Host-to-IMP

          return from FIRE
```

8.2.3.4.2 Not link Ø

Messages from terminals (i.e., messages not using link Ø) are sent from the TIP to the IMP by a set of routines. These are SENDIT, SENDW, SENDOF, MOVWRD, GETBUF, TOHOST, and TILEAD. The control structure of these routines is that SENDIT is called at clock level to lay claim to the TIP-to-IMP interface for a device. SENDW is called in the background loop and has a co-routine relationship to SENDOF. If a device has laid claim to the TIP-to-IMP interface, SENDW checks to see if the interface is not in use by the link Ø code, SENDAB. The interlock between the link Ø use of the interface and the data link use of the interface is the flag TIGOF. If SENDW can get the interface, it jumps into SENDOF. SENDOF does what it can about copying the data characters from the device input buffers to the IMP, debreaking via a call to SENDOF when it's necessary to wait for some reason. SENDOF calls TILEAD (also used by SENDAB) to give a leader to the IMP, calls GETBUF to get an IMP buffer to fill with data, calls MOVWRD to fill the buffer, and calls TOHOST to pass the filled buffer to the IMP.

SENDIT if some device has not already
got the TIP-to-IMP interface (DEVYCE ≠ Ø),
capture interface for current device
(DEVYCE←device)

make all the buffer visible

set the overrun flag

return from SENDIT

SENDW

if no device is giving stuff to the IMP (DEVYCE ≠ Ø), return from SENDW

if IMP is busy (TIGOF = -1), return from SENDW

if IMP is expecting beginning of message (TIGOF < -1), goto SENDL1

return from SENDOF (restart SENDOF where it last left off)

SENDOF return from SENDW

SENDL1 if link is blocked, return
from SENDW

if there is no message allocate,
return from SENDDW

subtract 1 from the message
allocate

if there is no bit allocate,
goto BFAIL2

clear OVERRUN flag (MDOVER←Ø)

call SHRINK to update character count

if there is nothing in the
input buffer to send, goto BFAIL2

if bit allocate is big enough for all
the characters to be sent,
goto SENDL8

SENDL9 set overrun flag so rest
will be sent later

if characters to be sent do not
fit in one message, goto SENDL1a

call TILEAD to send leader to IMP

build on extended header in IMP buffer

call MOVWRD repeatedly with successive characters from input buffer

call MOVWRD with padding

call TOHOST to close off message (last packet of message)

update bit allocate

return from SENDW

SENDL1a set number of characters to be sent to exactly 1 message

goto SENDL9

BFAIL2 add 1 back into message allocate

mark that device no longer using SENDOF (DEVYCE←Ø)

call SENDOF

BFAIL return form SENDW

```
MOVWRD      save the word

            if the IMP buffer is full,
            goto MOVWRDa

MOVWRDb     store the saved word
            in the buffer

            increment the buffer fill pointer

            return from MOVWRD



MOVWRDa     call TOHOST to give buffer to IMP
            (not last packet in message)

            call GETBUF to wait until IMP has a
            new buffer ready

            goto MOVWRDb
```

```
GETBUF      if IMP is not busy (i.e., has put up
            another buffer), return from GETBUF

            call SENDOF to wait for a new
            buffer

            return from GETBUF
```

```
TOHOST      set the last packet flag
            as appropriate

             et the blocked link bit

            fake interrupt to Host-to-IMP

            return from TOHOST
```

8.2.3.5 Accept Data From IMP (SLURP)

The routine SLURP is used to take messages from the IMP and to process them. The messages are of two main classes, irregular (IMP-to-Host control) messages which are processed by the routine IRREG, and regular messages. The regular messages in turn are of two classes, control messages on link Ø which are processed by the routine PROT, and data messages on links 3 through 65 which are processed by the routine DATABF. Of course various illegal messages may also arrive; these are discarded and reported as appropriate. The subroutine SDIS is used to sort out the message type and dispatch to the proper processing routine. The linkage between the rest of the background routines and the routines which will be described in this section is a co-routine linkage via the SLURP/SLURE co-routines.

SLURP if the IMP has no data for the TIP, return from SLURP

return from SLURE (to restart SLURE where it last left off)

SLURE save where left off for next call to SLURP

return from SLURP

```
FLUSHa    call NEXTBF to get next packet
          in message

FLUSH     if this is not last packet in message,
          goto FLUSHa

          call NEXTBF to get first
          packet in next message

          call SDIS to dispatch on
          message type

          call SLURE to pass control to other
          background routines

          goto DATABF
```

NEXTBF call H3OUIL to fake interrupt to IMP-to-Host routine

if the IMP has something for the TIP, return from NEXTBF

call SLURE to pass control to other background routines while waiting for something from IMP

calls SDIS to dispatch on message type

goto DATABF

```
SDIS        save link (in L2)

            save Host (in L1)

            if message is not a regular message
            from a real Host, goto IRREG

            save the Host/Host protocol byte count
            (in OUCNT1) and negative of byte count
            (in L4)

            if link number = 0/, goto PROT

            if link number = 1 or >65., goto BUGFA

            otherwise, 3≤ link ≤65 (or, 1≤
            device ≤63)

            if message is not from Host from
            which it is expected, report error and
            goto FLUSH

            dispatch to magtape option if appropriate

            return from SDIS
```

```
BUGFA     if link ≠ 66, report an error

          goto FLUSH
```

8.2.3.5.1 Irregular Messages

The irregular messages the TIP receives from the IMP are handled by the routine IRREG. The handling of the irregular messages is fairly straightforward, as shown by the following detailed logic descriptions. One complication is the handling of RFNMs and INCOMPLETE TRANSMISSIONs which gets involved with Host/Host protocol allocates and the TIP's terminal input buffering logic.

IRREG dispatch on message type

if type >10., =3, or =6 (undefined or NOP), goto FLUSH

if type =10. (INTERFACE RESET), report error and goto FLUSH

if type =1 (ERROR WITHOUT ID), report error and goto FLUSH

if type =8 (ERROR WITH ID), report error and goto FLUSH

if type =2 (IMP GOING DOWN), goto IGD

if type =5 (RFNM), goto IRFNM

if type =7 (DESTINATION DEAD), goto DEAD

otherwise type =9 (INCOMPLETE TRANSMISSION)

report "INCOMPLETE TRANSMISSION"

call UNBLK to unblock link

if link is already unblocked, goto FLUSH

if link is control link, goto IRINC3

IRINC1 scan through all devices looking for connections to Host which sent INCOMPLETE TRANSMISSION and marking allocates to be retransmitted as necessary--finally, poke allocate sender

goto FLUSH

IRINC3 restore allocates for Host to what
they were before last message was sent

goto IRFNM3

IGD loop through all devices setting
IMP-GOING-DOWN error message bit

goto FLUSH

IRFNM call UNBLK to unblock link

if link was already unblocked, goto FLUSH

if link is control link, goto IRFNM1

flush the RFNMed characters from the input buffer

if no characters are left in input buffer, goto IRFNM4

relocate them to bottom of the input buffer

IRFNM4 adjust input buffer pointer so it points the beginning of the available space

IRFNM3 call SENDIT and SENDW to attempt to send any new stuff to be sent

goto FLUSH

```
DEAD    call UNBLK to unblock link
        if link is control link, goto DEAD2
        call SHUTDN to shut down one connection
        call SHUTDN to shut down other connection
        mark PSTATE to close one connection
        mark QSTATE to close other connection
        goto DEAD3

DEAD2   if nobody is using logger, goto FLUSH
        if not talking to dead guy, goto FLUSH
        clear logger
DEAD3   tell user "DEAD"
        goto FLUSH
```

```
UNBLK       if link ≠ Ø, goto UNBLKa

            unblock the control link to this Host

            if link was not already unblocked, return
            from UNBLK

            report "EXTRA RFNM"

            return from UNBLK


UNBLKa      if link is 77 and if mag tape option, return
            from UNBLK

UNBLKb      search all devices for one using this link
            to this Host

            if done, goto UNBLKc

            if link not blocked, goto UNBLKb

            unblock link

            return from UNBLK


UNBLKc      report "EXTRA RFNM" error

            goto FLUSH
```

8 .2.3.5.2 Regular Messages

Regular messages come from the IMP in two forms, those on link zero (the Host/Host protocol control link) and those on data links (data messages for TIP terminals). The link Ø messages are processed by the routine PROT, and the data link messages are processed by the routine DATABF.

8.2.3.5.2.1 Link Ø Messages

Link Ø is used for Host/Host protocol control messages. The handling of these Host/Host protocol control messages is done in five pieces:

- handling RFCs and CLSs
- handling ECOs and RSTs
- handling ALLOCATES
- counting INSs
- ignoring illegal commands and commands not handled by the TIP.

The logic for dispatching on the various commands and ignoring the illegal and unhandled commands follows:

PROT	if no commands in the message, goto NOPROT
	set up GET1 co-routine to restart at GETNOP
PROTa	get next byte in message
	return from GET1 (restart GET1 where it last left off)
GET1	if there are any bytes left in message, goto PROTa
GETOUT	Poke PROBHO
	goto FLUSH
NOPROT	report reception of zero length protocol message
	goto FLUSH

```
GETNOPa   call GET2 and GET1 to flush bytes
          to the end of command

          call GET to get next character

GETNOP    dispatch on Host/Host protocol command type

          if ERR, RET, GVB, INR, ERP, RRP
          or NOP, goto GETNOPA

          if RTS, goto GETRTS

          if STR, goto GETSTR

          if CLS, goto GETCLS

          if ALL, goto GETALL

          if INS, goto GETINS

          if ECO, goto GETECO

          if RST, goto GETRST
```

8.2.3.5.2.1.1 Handling RFCs and CLSs

RFCs (Request for Connections) are handled by the GETRTC/GETSTR routine. CLSs (Closes) are handled by the GETCLS routine. RFCs and CLSs are driven by an eight-state finite-state machine for each device. The states are:

Ø -- try to open
1 -- RFC sent
2 -- RFC received, try to reply
3 -- connection open
4 -- try to close
5 -- CLS sent
6 -- CLS received, try to reply
7 -- no connection

External events drive the finite-state machine into its even-numbered state. Every time the TIP finds a device in an even-numbered state, it immediately performs the function which allows it to move the device to the next sequential odd-numbered state.

```
GETRTS
GETSTR  call SUCKS to get the socket numbers

        save (in EXTRA) the link or byte size,
        depending on whether RTS or STR

        call TESTOK to see if it's ok to act on RTS
        or STR; if not ok, goto TOOBAD

        if the finite state machine for this
        connection is in states 2,3,4,5, or 6,
        report error and goto GETNOP

        if state is Ø or 7, make state 1 and goto GETSTRa

        if state is 1, make state 2

GETSTRa save first socket number (in
        SOCKS1 and SOCKS2)

        if command is RTS, goto RTS
```

```
STR       set flag to send allocates to Host

          complete bit allocate

          poke PENALL to cause allocate
          to be sent

          save Host in HOSTR

          goto GETSTRb


RTS       save link and Host in HOSTS

GETSTRb   call ERRTEL to print "OPEN"

          poke PROBHO

          goto GETNOP
```

```
TOOBAD  if first "slot" in CLS sender
        is busy, goto TOOBADb

TOOBADa build a CLS in free slot

        goto GETEC1


TOOBADb if other "slot" is busy, goto GETNOP

        goto TOOBADa
```

```
TESTOK      if this device is looking
            for news, goto TE31

            if gender of connection is
            send, goto TE21

            if this device is not logging,
            error return from TOOBAD

            mark device to log

            goto GETNOP


TE21        if Host is wild, goto TE21a

            if Host does not match what
            expected, error return from TESTOK

TE21a       if socket is wild, good return
            from TESTOK

            if socket is not what's expected, error
            return from TESTOK

            good return from TESTOK


TE31        if socket is news socket,
            error return from TESTOK

            save Host, link, and sockets
            with device

            clear news bit

                                                  8/73
            good return from TESTOK
```

```
GETCLS  call SUCKS to get socket numbers from message

        call TESTOK to check if everything ok
        to start closing this procedure

        if not ok, goto GETNOP

        if connection is in state 0, 6, or 7,
        goto GETNOP

        if connection is in state 5 or 1,
        goto GETCLSa

        connection is in state 2, 3, or 4

        put connection in CLS
        received state

        goto STR4

GETCLSa call SHUTDN to clean
        things up

        put connection in closed state

        goto STR4
```

The following routine, SHUTDN, is called by GETCLS to clear up things associated with a closed connection. The routine is also called when a connection is closed because the Host went dead (by DEAD), because the Host sent a Reset (by SNDRP), and the Host refuses the ICP (by PROBCK). Consequently, the inclusion of SHUTDN under GETCLS is a little unnatural; however, under GETCLS is as natural as any other place.

```
SHUTDN  if device is wild, reset sockets
        and Host to "<any>"

        if shutting down send side, clear allocate
        counters and reset send link to Ø

        if not shutting down send link, mark to
        not retransmit allocate

        if device set to non-permanent
        @I N, reset escape to "@"

        clear INS/DM count

        return from SHUTDN
```

8.2.3.5.2.1.2 Handling ECOs and RSTs

As explained earlier, the TIP handles Host/Host protocol ECO and RST on a "catch as catch can" basis.

```
GETECH  call GET1 to flush byte to be echoed

        pickup echo-reply bit

        goto NEEDRP


GETRST  pickup Reset-reply bit

NEEDRP  OR in bit to proper flag by Host if not
        already set

        poke PENOTH flag to cause ERP or
        RRP to be sent

        goto GETNOP
```

8.2.3.5.2.1.3 Handling ALLOCATES

Received allocates are processed by the routine GETALL.

GETALL save link (in ALINK)

save message allocate (in AM)

save bit allocate (in AH and AL)

save word allocate (in ALL)

if there is no device from this Host using this link, goto GETNOP

if connection for this device is not in state 2, 3, 4, or 5, goto GETNOP

call mag tape if appropriate

if bit allocate $\leq 2^{16}$, goto FIND5

FIND6 mark bit allocation as infinite

goto FIND8

FIND5 if newly received bit allocate will cause bit allocation to overflow, goto FIND6

add newly received bit allocate to bit allocation

add newly received message allocate to message allocation

call SENDIT and SENDW to send allocate if possible

goto FLUSH

8.2.3.5.2.1.4 Counting INSs

Counting INSs is done by the routine GETINS in a completely straightforward manner.

GETINS if not $3 \leq link \leq 65$, goto GETNOP

increment the INS count for the device indicated by the link

goto GETNOP

8.2.3.5.2.2 Messages Not On Link Ø

Arriving messages which are not on link Ø contain data for output to devices. After a few checks for the proper format of the received messages, the data in the message is copied into the device's output buffer and, if necessary, output to the device is started.

DATABFa get device number of device being diverted to

store device number in OUDEV

DATABF saved device number in OUDEV

if device is diverting output, goto DATABFa

if there is no output buffer for this device, goto FLUSH

save max allocate which can be sent (in MAXALL)

if more data has arrived than device's output buffer can hold, goto DATABFb

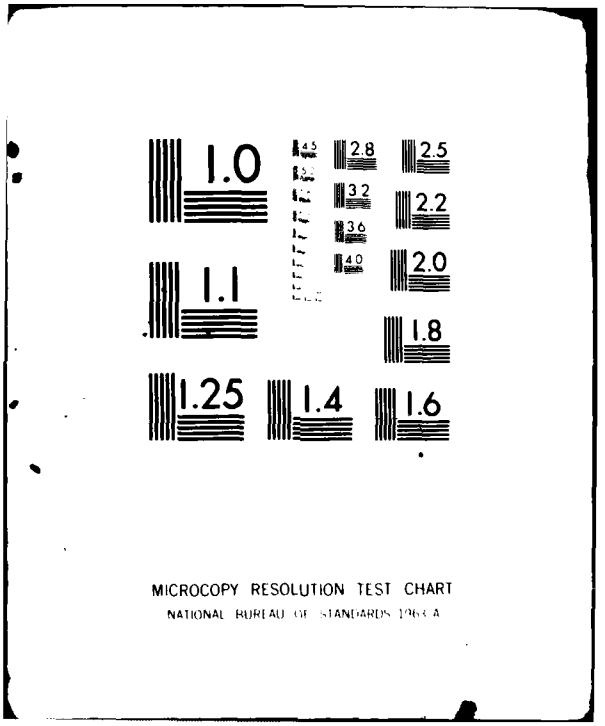
set up to send allocation equal to data received

DATABFc if no data in message received, goto NODATA

Copy bytes of received message into output buffer for device until message end or packet end, whichever comes first. If packet end comes first, call NEXTBF to get next packet of message and continue copy. When message ends, set mark to indicate there is output for device (MIGOTO ←1)

call OUNEW to start output to device

goto FLUSH


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

DATABFb report error

set up to send max allocation

goto DATABFc

8.2.3.5.2.2.1 OUNEW

The following routine, OUNEW, is used to start output to a device, if it is not presently doing any and there is something new to go to the device. The routine is also called by MODEMC and consequently its description here under DATABF is not completely natural; however, here seems as natural as any other place.

OUNEW

if output is in progress,
return from OUNEW

mark device active

build device table consisting of
a single device #: the terminal being
started up, in the table set the "fake
OI" bit (the sign bit).

lock interrupts & change PRIM

initialize # of "extra" OIs this device
requires to achieve full speed [by
calling IMAX]

call OI

restore PRIM

return from OUNEW

8.2.3.6 Modem and LIU Control

MODEM C — if we have set up no line to interrogate, goto MODEMa

if this is a "lethargic" pass, goto MODEMg [i.e. if DSFLAG is set]

read LIU status

if carrier not up, goto MODEMb

mark that carrier is present (by clearing MOCARR)

MODEM 8 — clear MOHANG

goto MODEMa

MODEMb — re-read LIU status

if data set ready is up, goto MODEMa

reset delay timer

if carrier was not up last time, goto MODEMa

mark that carrier is down (set MOCARR)

MODEMh — set to drop data terminal ready

call RESET to get the port unwound

goto MODEM8

MODEMa step to next device

if next device is not Ø, go to MODEMi

reset DSFLAG

if lethargic pass timer [DSCLK] has not run out, goto MODEMi

reset timer to one minute

set DSFLAG

MODEMi if output in progress, goto MODEMc

if MLC input controller is to be reset, goto MODEMd

if MLC output controller is to be reset, goto MODEMe

restart output with call to OUNEW

mark to raise data terminal ready next time

goto MODEMc

MODEMd get new input rate

goto MODEMf

MODEMe get new output rate

set "OI expected"

goto MODEMf

MODEMc get current state of data terminal ready

MODEMf output to MLC controller

return from MODEMc

MODEMg read LIU status

if data set is not in the state of carrier data set low and data set ready high, goto MODEM8

complement MOHANG

if MOHANG is now Ø, goto MODEMh

goto MODEMa

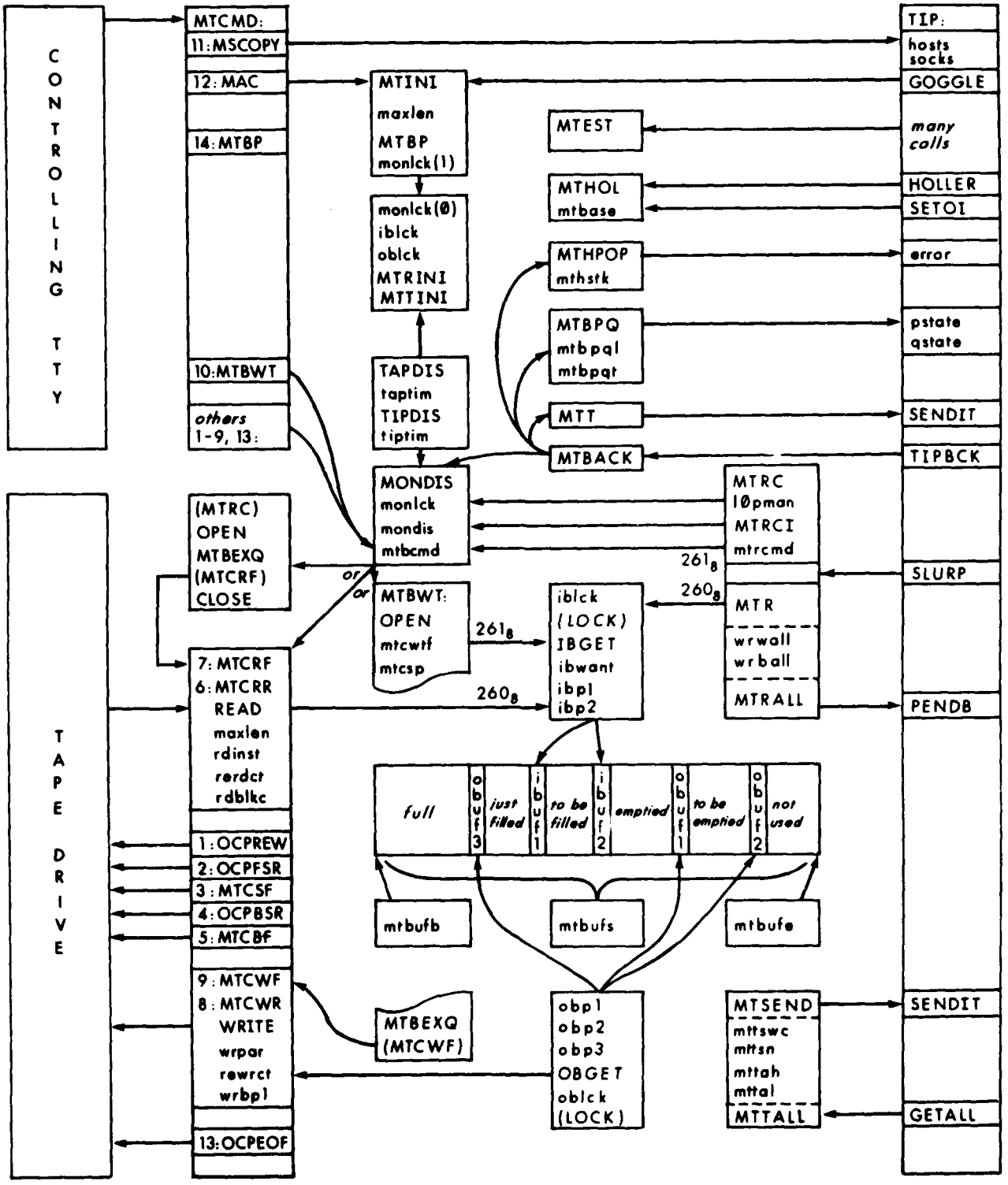
8.2.4 MLC Output Interrupt Routine

The routine which services the MLC Output Interrupt is called TOUT. In addition to being called by the MLC Output Interrupt, TOUT is also called by the Clock Interrupt routine (at OOPS) to restart MLC output when necessary.

```
TOUT    save AC

        save keys

        if there is not output to
        go, goto TOUTa

        set up an output from
        the buffer with stuff in it

        switch to other buffer for
        filling with more characters

        do output

TOUTa   restore keys

        restore  AC

        enable interrupts

        return
```

8.2.5 Magnetic tape option

There follows a block diagram for the TIP magnetic tape option. The magnetic tape option listing, attachment IV, should be studied along with the diagram.


CONTROLLING TTY
TAPE DRIVE
MTCMD:
11:MSCOPY
12:MAC
14:MTBP
10:MTBWT
others 1-9, 13:
7:MTCRF
6:MTCRR
READ
maxlen
rdinst
rerdct
rdblkc
1:OCPREW
2:OCPFSR
3:MTCSF
4:OCPBSR
5:MTCBF
9:MTCWF
8:MTCWR
WRITE
wrpar
rewrct
wrbpl
13:OCPEOF
MTINI
maxlen
MTBP
monlck(1)
monlck(0)
iblck
oblck
MTRINI
MTTINI
TAPDIS
taptim
TIPDIS
tiptim
MONDIS
monlck
mondis
mtbcmd
(MTRC)
OPEN
MTBEXQ
(MTCRF)
CLOSE
MTBWT:
OPEN
mtcwtf
mtcsp
MTBEXQ
(MTCWF)
MTEST
MTHOL
mtbase
MTHPOP
mthstk
MTBPQ
mtbpql
mtbpqt
MTT
MTBACK
iblck
(LOCK)
IBGET
ibwant
ibp1
ibp2
261₈
260₈
MTRC
10pman
MTRCI
mtrcmd
MTR
wrwall
wrball
MTRALL
full
obuf3
just filled
ibuf1
to be filled
ibuf2
emptied
obuf1
to be emptied
obuf2
not used
mtbufb
mtbufs
mtbufe
obp1
obp2
obp3
OBGET
oblck
(LOCK)
MTSEND
mttswc
mttsn
mttah
mttal
MTTALL
TIP:
hosts
socks
GOGGLE
many calls
HOLLER
SETOI
error
pstate
qstate
SENDIT
TIPBCK
SLURP
PENDB
SENDIT
GETALL

8.3 Index to Detailed Descriptions

ERRTELA	8.2.2.2.4-6
ERRTELB	8.2.2.2.4-6
ESC	8.2.2.1.1-7
ESC1	8.2.2.1.3.1-2
ESC1A	8.2.2.1.3.1-3
ESC2	8.2.2.1.1-7
EXIT	8.2.2.1.3.1-4
FASTER	8.2.2.2.1-4
FEED	8.2.2.1.1-6
FIND5	8.2.3.5.2.1.3-2
FIND6	8.2.3.5.2.1.3-2
FIRE	8.2.3.4.1-2
FIXECH	8.2.2.2.2.1-2
FLUSH	8.2.3.5-3
FLUSHA	8.2.3.5-3
GENPAR	8.2.2.2.5.3-1
GET1	8.2.3.5.2.1-2
GETALL	8.2.3.5.2.1.3-1
GETBUF	8.2.3.4.2-7
GETCLS	8.2.3.5.2.1.1-6
GETCLSA	8.2.3.5.2.1.1-6
GETECH	8.2.3.5.2.1.2-1
GETINS	8.2.3.5.2.1.4-1
GETNOP	8.2.3.5.2.1-3
GETNOPA	8.2.3.5.2.1-3
GETOUT	8.2.3.5.2.1-2
GETRST	8.2.3.5.2.1.2-1
GETRTS	8.2.3.5.2.1.1-2
GETSTR	8.2.3.5.2.1.1-2
GETSTRA	8.2.3.5.2.1.1-2
GETSTRB	8.2.3.5.2.1.1-3
GOGGLE	8.2.1-2
HOLL2	8.2.2.2.4-5
HOLLER	8.2.2.2.4-5
HUNT	8.2.2.1.4-2
HUNTA	8.2.2.1.4-2
IBMCON	8.2.2.1.1-4
IBMECO	8.2.2.1.1-4
IBMEEE	8.2.2.1.1-4
IBMESC	8.2.2.1.1-4
IBMIN	8.2.2.1.5.1-2
IBMOUT	8.2.2.2.5.1-2
IBMQ1	8.2.2.1.5.1-2
IBMQ10	8.2.2.1.5.1-3
IBMQ2	8.2.2.1.5.1-2
IBMQ3	8.2.2.1.5.1-3
IBMQ4	8.2.2.1.5.1-3
IBMQ5	8.2.2.1.5.1-3
IBMQ6	8.2.2.1.5.1-4

IBMQ7	8.2.2.1.5.1-5
IBMQ8	8.2.2.1.5.1-5
IBMQ9	8.2.2.1.5.1-2
IGD	8.2.3.5.1-4
INBS	8.2.2.1.5.1-8
INBS2	8.2.2.1.5.1-8
INCC	8.2.2.1.5.1-9
INC2	8.2.2.1.5.1-9
INC4	8.2.2.1.5.1-9
INDQ	8.2.2.1.5.1-7
INLC	8.2.2.1.5.1-5
INLF	8.2.2.1.5.1-6
INNL	8.2.2.1.5.1-9
INTAB	8.2.2.1.5.1-10
INUC	8.2.2.1.5.1-5
INUCA	8.2.2.1.5.1-5
IRFNM	8.2.3.5.1-5
IRFNM3	8.2.3.5.1-5
IRFNM4	8.2.3.5.1-5
IRINC1	8.2.3.5.1-2
IRINC3	8.2.3.5.1-3
IRREG	8.2.3.5.1-2
LAST8	8.2.2.2-2
LCHAR	8.2.2.2-2
LCHAR3	8.2.2.2-2
LFCHR	8.2.2.1.3.1-7
LFCHRA	8.2.2.1.3.1-7
LINBSY	8.2.2.2.1-3
LINBSYA	8.2.2.2.1-3
MEM4	8.2.2.2.5.4.1-3
MEMA1	8.2.2.2.5.4.1-3
MEMNUL	8.2.2.2.5.4.1-2
MEMRX	8.2.2.2.5.4.1-2
MEMRXA	8.2.2.2.5.4.1-2
MEMRXB	8.2.2.2.5.4.1-3
MEMRXC	8.2.2.2.5.4.1-3
MEMSVI	8.2.2.2.5.4.1-2
MIFAST	8.2.2.2.1-4
MODEM8	8.2.3.6-1
MODEMA	8.2.3.6-2
MODEMB	8.2.3.6-1
MODEMC	8.2.3.6-1
MODEMC	8.2.3.6-3
MODEMD	8.2.3.6-2
MODEME	8.2.3.6-3
MODEMF	8.2.3.6-3
MODEMG	8.2.3.6-3
MODEMH	8.2.3.6-1
MODEMI	8.2.3.6-2

MOVWRD	8.2.3.4.2-6
MOVWRDA	8.2.3.4.2-6
MOVWRDB	8.2.3.4.2-6
NCH2	8.2.2.2-5
NCH2A	8.2.2.2-6
NCH3	8.2.2.2-6
NCHAR	8.2.2.2-5
NCHC	8.2.2.2-4
NEEDRP	8.2.3.5.2.1.2-1
NETCOM	8.2.2.2.2-3
NEWBUF	8.2.2.2-3
NEWBUFA	8.2.2.2-3
NEWBUFB	8.2.2.2-3
NEXTBF	8.2.3.5-4
NOPE	8.2.2.1.1-5
NOPROT	8.2.3.5.2.1-2
OI	8.2.2.2-1
OIBS	8.2.2.2.5.1-7
OIBSA	8.2.2.2.5.1-7
OIBSB	8.2.2.2.5.1-7
OIL1	8.2.2.2-6
OIL2	8.2.2.2-1
OILF	8.2.2.2.5.1-8
OITAB	8.2.2.2.5.1-9
OOPS	8.2.2-2
OUNEW	8.2.3.5.2.2.1-2
P01	8.2.2.2.5.1-4
P01A	8.2.2.2.5.1-6
P02	8.2.2.2.5.1-2
P03	8.2.2.2.5.1-5
P04	8.2.2.2.5.1-2
P05	8.2.2.2.5.1-5
P06	8.2.2.2.5.1-6
P09	8.2.2.2.5.1-3
P10	8.2.2.2.5.1-3
P11	8.2.2.2.5.1-4
P13	8.2.2.2.5.1-2
P14	8.2.2.2.5.1-5
P15	8.2.2.2.5.1-6
P15A	8.2.2.2.5.1-6
P15B	8.2.2.2.5.1-6
P17	8.2.2.2.5.1-5
PAR	8.2.2.1.3.1-6
PEND1	8.2.3.1-2
PEND3	8.2.3.1-2
PEND4	8.2.3.1-2
PEND5	8.2.3.1-2

PEND6	8.2.3.1-2
PEND7	8.2.3.1-3
PEND8	8.2.3.1-2
PENDA	8.2.3.1.1.1-1
PENDB	8.2.3.1.1.2-1
PENDB1	8.2.3.1.1.2-1
PENDB2	8.2.3.1.1.2-2
PENDC	8.2.3.1.2.1-1
PENDD	8.2.3.1.2.2-1
PENDE	8.2.3.1.1.3-1
PENDH	8.2.3.1.2.2-1
PENDIN	8.2.3.1-2
PENDUN	8.2.3.1.2.3-1
PRO3	8.2.3.2-3
PRO3A	8.2.3.2-3
PRO5	8.2.3.2-3
PRO5A	8.2.3.2-3
PROB1	8.2.3.2-2
PROB2	8.2.3.2-2
PROB3	8.2.3.2-2
PROBCK	8.2.3.2-2
PROT	8.2.3.5.2.1-2
PROTA	8.2.3.5.2.1-2
Q2	8.2.2.1.3.1-4
Q3	8.2.2.1.3.1-6
Q3A	8.2.2.1.3.1-6
Q6	8.2.2.1.3.1-7
Q8	8.2.2.1.3.1-8
Q9	8.2.2.1.3.1-8
QBA	8.2.2.1.3.1-7
REG	8.2.2.1.1-5
REGA	8.2.2.1.1-5
RESET	8.2.1-4
RESET5	8.2.1-4
RESETA	8.2.1-4
RESMOR	8.2.2.2.1-4
RETURN	8.2.2.1.3.1-5
RETURNA	8.2.2.1.3.1-5
RTS	8.2.3.5.2.1.1-3
SDIS	8.2.3.5-5
SEND6	8.2.3.4.1-1
SENDAB	8.2.3.4.1-1
SENDABA	8.2.3.4.1-2
SENDABB	8.2.3.4.1-1
SENDIT	8.2.3.4.2-2
SENDL1	8.2.3.4.2-4
SENDL1A	8.2.3.4.2-5

SENDL9	8.2.3.4.2-4
SENDOF	8.2.3.4.2-4
SENDW	8.2.3.4.2-3
SET6	8.2.2.2.4-2
SETOI	8.2.2.2.4-2
SETOI2	8.2.2.2.4-3
SETOIA	8.2.2.2.4-2
SETOIB	8.2.2.2.4-2
SETOIC	8.2.2.2.4-3
SFUDGE	8.2.2.2.2-1
SHUTDN	8.2.3.5.2.1.1-8
SLOWCR	8.2.2.2.5.2-1
SLOWCRA	8.2.2.2.5.2-2
SLOWCRB	8.2.2.2.5.2-1
SLOWCRC	8.2.2.2.5.2-1
SLOWCRD	8.2.2.2.5.2-2
SLOWCRE	8.2.2.2.5.2-2
SLURE	8.2.3.5-2
SLURP	8.2.3.5-2
SNDRP	8.2.3.1.2.2-2
SNDRPA	8.2.3.1.2.2-2
SPBYDM	8.2.2.2.2-1
SPBYTE	8.2.2.2.2-1
SPBYTEA	8.2.2.2.2-1
SPBYTEC	8.2.2.2.2-1
SPIAC	8.2.2.2.2-2
SPIACA	8.2.2.2.2-2
SSYN1	8.2.2.1.3.1-6
STR	8.2.3.5.2.1.1-3
SYNTAX	8.2.2.1.3.1-7
TE21	8.2.3.5.2.1.1-5
TE21A	8.2.3.5.2.1.1-5
TE31	8.2.3.5.2.1.1-5
TESTOK	8.2.3.5.2.1.1-5
TILEAD	8.2.3.4-1
TOHOST	8.2.3.4.2-8
TOOBAD	8.2.3.5.2.1.1-4
TOOBADA	8.2.3.5.2.1.1-4
TOOBADB	8.2.3.5.2.1.1-4
TOUT	8.2.4-2
TOUTA	8.2.4-2
UNBLK	8.2.3.5.1-7
UNBLKA	8.2.3.5.1-7
UNBLKB	8.2.3.5.1-7
UNBLKC	8.2.3.5.1-7
WILL	8.2.2.2.2-2

ATTACHMENTS

I.	Macros	MACROS,46,PARAMS
II.	Concordance	TIPCON,316,TIP
III.	Listing	TIPLST,316,TIP
IV.	Mag tape option	MAGLST,1,MAGTAP


FILMED
8